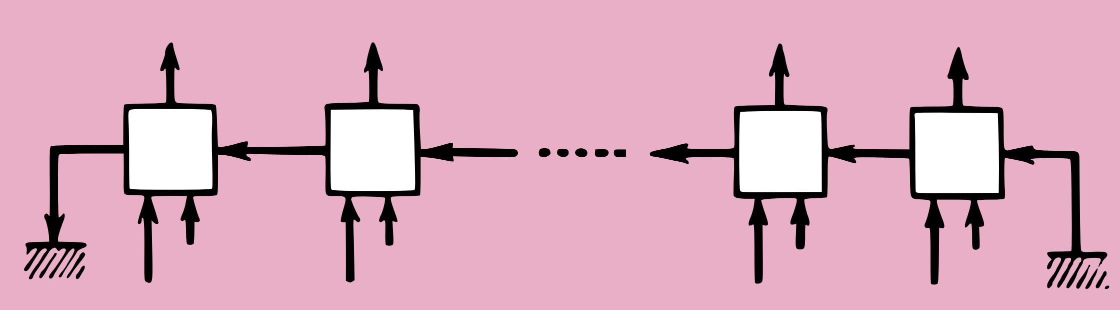
N. Krinitski, G. Mironov, C. Frolov

PROGRAMMATION ET LANGAGES SYMBOLIQUES



Éditions Mir Moscou

Н. А. КРИНИЦКИЙ, Г. А. МИРОНОВ, Г. Д. ФРОЛОВ

ПРОГРАММИРОВАНИЕ И АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ

Под редакцией А. А. ДОРОДНИЦЫНА

ИЗДАТЕЛЬСТВО «НАУКА»
МОСКВА

N. KRINITSKI, G. MIRONOV, G. FROLOV

PROGRAMMATION ET LANGAGES SYMBOLIQUES

Sous la direction de
A. DORODNITSYNE,
de l'Académie des Sciences de l'U.R.S.S.

ÉDITIONS MIR · MOSCOU

Traduit du russe
par V. KHARINE

На французском языке

EXTRAIT DE LA PRÉFACE À L'ÉDITION RUSSE

En moins de 20 ans, trois générations de calculateurs se sont succédé. Les calculateurs de deux premières générations ont cédé la place aux machines de la troisième génération caractérisées par l'emploi de circuits intégrés à semiconducteurs et surtout par l'évolution de la notion de software sans lequel leur utilisation est impossible.

La résolution de problèmes proposés à un calculateur n'est possible que lorsque la mémoire de celui-ci contient, en plus d'un programme de résolution, des programmes spéciaux formant ce qu'on appelle système d'exploitation. Considérée avec ces programmes, la machine initiale représente un nouveau calculateur, qui diffère de la machine de base par des particularités de fonctionnement et par le langage d'entrée.

Les systèmes d'exploitation permettent le traitement en multiprogrammation ainsi que l'utilisation de la machine en temps partagé par un grand nombre d'utilisateurs. Muni d'un système d'exploitation convenable, un calculateur devient une véritable machine de gestion.

En plus du système d'exploitation, le software d'un calculateur comprend des programmes traducteurs qui assurent la traduction automatique des programmes, rédigés dans des langages algorithmiques convenables, dans le langage machine. Il est clair que le programmeur n'a plus besoin de connaître en détail les instructions de la machine.

Jusqu'à ces derniers temps, le développement de la théorie et des méthodes de programmation était tellement sinueux que chaque nouvelle étape annulait, dans une grande mesure, les acquisitions des étapes précédentes. A l'heure actuelle la théorie de programmation semble avoir trouvé une voie qu'elle suivra longtemps, tout en s'enrichissant et évoluant. Le développement ultérieur des calculateurs donnera lieu à de nouveaux chapitres de la théorie sans annuler les précédents.

En rédigeant le présent livre, les auteurs se sont appuyés sur les acquisitions de la nouvelle étape du développement de la théorie

de programmation. On y expose d'abord les fondements de la théorie de programmation: des éléments de logique mathématique, de théorie des langages formels et de théorie des algorithmes. Toutes les autres parties du livre se réfèrent à ces notions de base. Les calculateurs sont envisagés comme des réalisations physiques des algorithmes d'exécution des programmes; les programmes eux-mêmes — comme des classes spéciales d'algorithmes; les codes d'instructions de machine — comme des langages algorithmiques qu'on appelle langages machine. Un chapitre du livre est consacré à la notion fondamentale de software. La plus grande partie du livre est consacrée à la description des langages symboliques les plus usités dont le YALS (langage des schémas logiques) qui convient particulièrement pour les transformations équivalentes des algorithmes et qu'on peut utiliser en tant que langage intermédiaire dans les travaux d'automatisation des processus et pour la programmation dans un langage d'assemblage. Les langages d'assemblage et les langages ALGOL 60, FORTRAN, PL/1 et COBOL sont des langages d'entrée de la programmation moderne. Le YALS est décrit le premier, puisqu'il est utilisé pour la description des langages machine et les transformations équivalentes des algorithmes.

Les auteurs ont considéré utile de retenir des éditions précédentes du livre certaines connaissances sur la programmation dans les langages machine, ceci pour les raisons suivantes: premièrement, chaque programmeur doit comprendre les principes de fonctionnement des calculateurs, or ils sont impossibles à assimiler complètement sans une connaissance des bases de la programmation manuelle; deuxièmement, les notions mentionnées sont indispensables pour les équipes qui doivent s'occuper du software.

Les chapitres 1 à 5, 7 et 10 sont rédigés par N. A. Krinitski, le chapitre 6 par G. A. Mironov et les chapitres 8, 9 par G. D. Frolov.

Nous estimons qu'il est de notre devoir de rappeler que la présente approche de la théorie de programmation et que le contenu de ce livre sont grandement influencés par les œuvres et les idées de notre maître A. A. Liapounov, membre correspondant de l'Académie des sciences de l'U.R.S.S., que nous avons perdu en 1973.

Les auteurs
Mai 1975

PRÉFACE À L'ÉDITION FRANÇAISE

Le berceau de la programmation (en tant que problème et en tant que spécialité) en U.R.S.S. fut entouré de mathématiciens. Aujourd'hui encore, un grand nombre de mathématiciens consacrent leurs efforts à la théorie et à la pratique de la programmation.

C'est la raison pour laquelle la programmation devint d'emblée une science. Dès 1951-1953 A. A. Liapounov en exprima les idées fondamentales en analysant le processus de programmation, et en arrivant à la notion de schéma logique du programme, puis en formulant la notion d'opérateur de programme. C'est sur ce fondement que fut construite la méthode de programmation opératorielle *), que furent créés les premiers compilateurs soviétiques (on les appelait à l'époque « programmes de programmation »), écrits les premiers ouvrages sur les transformations équivalentes des algorithmes. En 1953, le mathématicien soviétique A. A. Markov proposa une théorie des algorithmes normaux qui, tout en appartenant à la logique mathématique, exerça une grande influence sur le développement des idées de la théorie de programmation.

Les programmeurs soviétiques disposent aujourd'hui des résultats théoriques et pratiques obtenus en U.R.S.S. et à l'étranger. Notre pays attache une grande importance à la formation des cadres. Non seulement les universités, mais aussi plusieurs écoles supérieures techniques forment maintenant des spécialistes en programmation. Dans ce but, on y crée des chaires, et parfois des sections de mathématiques appliquées.

D'après ce livre, on peut, dans une certaine mesure, se faire une idée sur le développement de la programmation en U.R.S.S., bien que, pour des raisons évidentes, les dernières études n'y soient pas présentées (les méthodes de programmation parallèle, les récents résultats des mathématiques appliquées et de la théorie non tradi-

*) Dans la traduction des chapitres 6 à 10 du livre consacrés aux principaux langages de programmation, le terme « opérateur de programme » est remplacé par le terme instruction, plus proche de la terminologie française. (Note du traducteur.)

tionnelle des algorithmes, orientés vers les calculateurs et la programmation).

Si le présent ouvrage intéresse le lecteur français, les auteurs espèrent lui faire connaître, dans un proche avenir, certaines matières qui n'y figurent pas.

Les auteurs profitent de l'occasion pour exprimer leur profonde reconnaissance à M. Kharine qui a réalisé la traduction du livre et en a revu le texte.

Les auteurs
Septembre 1977

CHAPITRE PREMIER

ÉLÉMENTS DE LA THÉORIE DE LA PROGRAMMATION

§ 1.1. Notions fondamentales

La théorie de la programmation et des calculateurs commandés par programmes est basée sur la théorie des langages formels, les éléments de la théorie des nombres, la logique mathématique, la théorie des algorithmes *), ainsi que sur la théorie des systèmes complexes, cette dernière dépassant le cadre des disciplines précédentes, bien qu'étant liée à elles.

Le présent chapitre n'embrasse qu'une partie de la théorie de la programmation; en particulier, on n'y traite point des matières s'appuyant sur la théorie des systèmes complexes.

1.1.1. Lettres, connecteurs, constructions. Dans ce qui suit, un grand rôle appartient à la notion de construction. On entend par construction une structure formée à partir d'éléments à l'aide de connecteurs. Les éléments les plus simples d'une construction (« matériel de construction ») sont des lettres ; les entités plus compliquées se composent elles-mêmes de quelques éléments à lier « cimentés » par des connecteurs. Voici les définitions rigoureuses d'une lettre, d'un connecteur et d'une construction.

On appelle *lettres* et *connecteurs* les entités que l'on considère comme indissolubles et invariantes, vérifiant les quatre conditions suivantes :

— De toute lettre (de tout connecteur) on sait à tout moment que c'est une lettre (un connecteur).

— Etant données deux lettres (deux connecteurs), on sait à tout moment si elles (ils) sont *identiques* ou *différentes*. (La reconnaissance de l'identité s'appelle comparaison.)

— N'importe quel nombre de connecteurs peuvent lier un élément commun ; en particulier, ce peut être une lettre.

— Un connecteur ne peut lier qu'un nombre bien déterminé d'éléments (ce nombre est appelé *rang* de connecteur), ou bien, comme on dit encore, il possède un nombre déterminé de *branches* (leur nombre est égal au rang); les branches d'un connecteur sont

*) La théorie des machines de Turing et celle des automates sont des parties de la théorie des algorithmes.

réparties en groupes classés par ordre de supériorité (il est commode d'attribuer à ces groupes les numéros successifs 1, 2, . . ., k; ceci fait, des deux groupes de branches on considère comme supérieur celui dont le numéro est plus grand). On appelle *genre* d'une branche de connecteur le numéro du groupe dont elle fait partie et *genre* du connecteur lui-même, le nombre de groupes de ses branches. On considère que les branches d'un même genre sont identiques, les branches de genres différents étant différentes. On appelle *caractéristique* d'un connecteur une suite de nombres $n_1, n_2, \ldots, n_k$, où n_i est la quantité de ses branches du genre i. La description complète d'un connecteur (d'une liaison) est donnée par son nom et sa caractéristique.

Signalons que les deux premières conditions portent sur les propriétés communes des lettres et des connecteurs, les deux dernières établissent les distinctions.

EXEMPLE. 1.1. Les lettres françaises sont, de notre point de vue, des *lettres*. Les lettres individuelles occupant la première et la troisième place dans le mot

a n a l y s e

sont identiques. On peut dire que ce sont des copies d'une même lettre.

Les connecteurs qui « réunissent » les lettres en un tout sont ici interprétés par une disposition particulière des letrres: deux lettres sont liées entre elles si elles sont placées l'une à côté de l'autre.

Supposons qu'il y ait un certain nombre de connecteurs et d'éléments à lier.

Une liaison est dite *saturée* si le nombre d'éléments qu'elle lie est égal à son rang.

Deux éléments donnés sont dits *directement liés* lorsqu'ils sont à lier par un même connecteur.

Deux éléments donnés sont dits *liés* lorsque ou bien a) ils sont directement liés, ou bien b) l'un d'eux est lié à un élément qui est directement lié à l'autre.

C'est pour la première fois que nous formulons une définition dite *récursive*. Le lecteur connaît, bien sûr, ce que c'est qu'une définition directe où la notion définie est introduite à l'aide d'une ou de plusieurs notions qui ne dépendent pas d'elle. En voici un exemple: « on appelle triangle une ligne brisée fermée qui se compose de trois segments ». Les « définitions » qu'on appelle cercles vicieux ne sont pas moins connues. Là, on exprime la notion à définir par d'autres notions parmi lesquelles il y a des notions exprimées par celle à définir. Voici un cercle vicieux: « on appelle point un segment dont la longueur est nulle, et on appelle segment une partie de droite

limitée par deux points ». En mathématique on évite les cercles vicieux.

Une définition récursive comprend les parties directe et cyclique. La dernière n'est pourtant pas un cercle vicieux, puisque, par la partie directe, la notion à définir reçoit une signification initiale qui s'élargit en appliquant plusieurs fois la partie cyclique. Dans la définition ci-donnée d'éléments liés la partie directe est présentée par le point a) et la partie cyclique par le point b).

Une définition récursive peut se composer de plusieurs parties directes et cycliques, ces dernières étant susceptibles de « s'enchevêtrer » d'une manière fort compliquée.

Une définition directe de la notion d'éléments liés peut être formulée comme suit.

Deux éléments x et y choisis parmi les éléments donnés sont dits liés, si l'on peut former à partir des éléments donnés une suite finie dont chaque terme (sauf le dernier) soit directement lié à son suivant et dont le premier et le dernier termes coïncident respectivement avec x et y.

Dans ce cas la définition directe est beaucoup plus compliquée que la définition récursive formulée plus haut. Dans certains cas il est pratiquement impossible de remplacer une définition récursive par une définition directe.

Lorsque les êtres à lier (ou certains d'entre eux) sont complexes, i.e. formés à l'aide de connecteurs à partir d'êtres plus simples dont certains, à leur tour, peuvent s'avérer composés, on peut parler de *lien spécial* entre les termes d'une construction (seul le lien spécial direct nous intéressera, c'est pourquoi nous omettrons le mot « direct »). A propos d'un être composé isolé nous dirons qu'il est *lié spécialement* s'il y a une liaison saturée pour laquelle les éléments à lier sont l'être composé donné et l'un ou plusieurs éléments faisant partie de celui-ci (ou faisant partie de ses parties, ou faisant partie de parties de ses parties, etc.). Deux êtres seront dits liés spécialement si l'une des possibilités suivantes se réalise :

1) l'un de ces êtres est directement lié par une liaison à deux branches (au sens ordinaire) à un élément faisant partie du second être (cet élément pouvant être plongé aussi profondément que l'on veut dans le second) ;

2) des parties composantes des deux êtres (composés) sont directement liées au sens ordinaire par une liaison à deux branches.

Plusieurs êtres seront dits *liés spécialement*, s'il y a un connecteur qui lie deux de ces êtres spécialement, et deux quelconques de ces êtres sont soit liés spécialement, soit directement (au sens ordinaire), le connecteur mentionné étant saturé.

Introduisons d'une manière récursive la notion de *construction.*

1. On appelle êtres (éléments) séparés soit les lettres, soit les constructions particulières sans partie commune.

2. Les constructions particulières représentent ou bien des constructions primaires, ou bien des constructions primaires dont certains éléments séparés sont liés spécialement.

3. On appelle construction primaire l'ensemble de plusieurs connecteurs et de plusieurs êtres séparés (contenant au moins un connecteur et au moins un être séparé), dont chaque connecteur est saturé et deux êtres quelconques séparés sont liés.

4. On appelle construction soit l'ensemble vide d'êtres séparés et de connecteurs, soit une construction particulière.

Les connecteurs mentionnés dans le point 3 de la définition d'une construction sont dits extérieurs par rapport aux constructions particulières liées et intérieurs par rapport à la construction. Un connecteur qui lie une construction particulière et un être qui appartient à cette construction ou à une autre construction particulière est dit semi-extérieur par rapport à chacune de ces constructions. Des exemples éclaircissant les notions introduites ici de lettres, connecteurs et constructions seront donnés dans les divisions suivantes.

Dans une construction, on peut se figurer chaque construction particulière, qui est un élément à lier et n'est pas une lettre séparée, comme contenue sous une enveloppe. Une branche de connecteur mène non pas à des parties de l'élément à lier (ceci conduirait à l'apparition d'un caractère supplémentaire d'une des parties), mais à l'enveloppe. Puisque la notion d'enveloppe ne figure pas dans la définition d'une construction, il faut préciser, pour chaque type de liens, quelles sont les constructions particulières susceptibles d'être liées par les branches de chaque genre.

1.1.2. Mots, alphabets, classification des constructions. Une classe des constructions qu'on appelle mots est d'un intérêt particulier.

On appelle *mot* une construction qui est soit vide, soit satisfait aux quatre conditions suivantes:

— toutes ses lettres sont liées moyennant les connecteurs de trois types: un connecteur *de début* de rang 1, un connecteur *de fin* de rang 1 et un connecteur *de prolongation* de rang 2, ce dernier ayant une caractéristique 1, 1;

— elle contient un et un seul connecteur de début, de même qu'un et un seul connecteur de fin;

— chaque lettre est susceptible d'être liée par deux et seulement deux connecteurs;

— si l'on pose que la branche du connecteur de début est de genre 1 et celle du connecteur de fin de genre 2, alors les branches des connecteurs liant chaque lettre du mot sont de genres différents.

La lettre d'un mot qui est liée par un connecteur de début s'appelle *début du mot.*

La lettre d'un mot qui est liée par un connecteur de fin s'appelle *fin du mot.*

Le nombre des lettres d'un mot s'appelle sa longueur. La longueur d'un mot vide est nulle.

Un mot vide n'a ni début ni fin.

En écrivant les mots (voir l'exemple 1.1) on n'explicite pas les connecteurs, on les interprète habituellement par la disposition relative des lettres sous forme d'une suite dont la lettre qui est le plus à gauche représente le début du mot, celle qui est le plus à droite, la fin du mot, parmi deux lettres voisines celle à gauche est précédente, celle à droite suivante (ces deux lettres sont liées par un connecteur de prolongation).

Pour limiter une classe de constructions on utilise le procédé suivant. On introduit un mot dont toutes les lettres sont deux à deux distinctes. Pris en ce sens, ce mot est appelé *alphabet*. Chaque lettre identique à l'une des lettres d'un alphabet A s'appelle *lettre dans* A. Chaque construction qui ne contient aucune lettre n'étant pas une lettre dans A s'appelle *construction dans* A. Ce procédé est suffisant pour borner la classe des mots (la collection des types de connecteurs utilisés pour former les mots étant donnée). En cas de constructions plus compliquées on introduit encore l'alphabet de connecteurs, c'est-à-dire un alphabet dont les lettres sont les symboles désignant les connecteurs (ou bien les symboles désignant les types de connecteurs, lorsque ces derniers sont représentés non par des symboles, mais par la disposition relative des éléments à lier). Tout connecteur identique à l'un des symboles (ou désigné par l'un des symboles) d'un alphabet de connecteurs B s'appelle *connecteur dans* B. Toute construction dans B qui ne contient aucun connecteur n'étant pas un connecteur dans A s'appelle *construction de la classe* (A, B).

Si chaque lettre d'un alphabet A est une lettre dans un alphabet B, on dit que B est *extension de* A et que A est une *partie* de B.

Deux alphabets sont dits égaux lorsque chacun d'eux est une partie de l'autre.

Le nombre des lettres d'un alphabet s'appelle sa *taille*. Si la taille d'un alphabet est n, alors évidemment le nombre d'alphabets différents qui sont égaux à l'alphabet donné sans coïncider avec lui est $n! - 1$.

Soient et A et B deux alphabets. Un alphabet constitué par toutes les lettres communes à A et B s'appelle *intersection* des alphabets A et B. L'intersection de deux alphabets égaux est égale à chacun d'eux.

Un alphabet qui est extension de l'alphabet A de même que de l'alphabet B et dont chaque lettre est une lettre dans A ou dans B s'appelle *réunion* (somme) des alphabets A et B. La réunion de deux alphabets égaux est égale à chacun d'eux.

Quels que soient deux alphabets A et B, il existe beaucoup d'alphabets qui sont leurs réunions (intersections); leur nombre est $n!$, où n est la taille d'une réunion (intersection). Dans certains

cas l'ordre des lettres dans un alphabet est essentiel. La somme des alphabets A et B qu'on peut obtenir en écrivant à la suite du mot A les lettres du mot B qui ne sont pas des lettres dans A, dans l'ordre dans lequel elles se suivent dans B, sera appelée somme principale des alphabets A et B et désignée

$$A \cup B.$$

Toute autre somme sera désignée par une nouvelle majuscule latine, par exemple C, $C = A \cup B$. D'une manière analogue nous appellerons intersection principale des alphabets A et B le mot qui s'obtient du mot A en y barrant toutes les lettres qui ne sont pas des lettres dans B. L'intersection principale des alphabets A et B sera désignée

$$A \cap B,$$

et pour désigner n'importe quelle autre intersection nous introduirons une nouvelle lettre.

Parfois nous écrirons un alphabet entre accolades, sous forme d'une suite des lettres séparées par des virgules.

EXEMPLE 1.2. Les alphabets

$$\{a, b, c, d\}$$

et

$$\{b, a, d, c\}$$

sont égaux entre eux.

Soient deux alphabets

$$\{1, b, 2, c, d\} \text{ et } \{1, b, c, 5\}.$$

Un exemple d'intersection de ces alphabets est

$$\{1, b, c\}$$

et un exemple de leur réunion est

$$\{1, b, 2, c, 5, d\}.$$

Deux mots sont dits égaux s'ils se composent de mêmes lettres disposées dans un même ordre.

Il est clair que chaque mot dans un alphabet A est un mot dans toute extension de l'alphabet A.

Les mots dans un alphabet doivent satisfaire à certaines conditions. Premièrement, pour n'importe quelles deux lettres d'un mot, on doit savoir quelle est la lettre précédente et quelle est la suivante. Deuxièmement, on doit connaître le début et la fin du mot. Enfin, il est parfois nécessaire de savoir décomposer un mot en lettres dans un alphabet A (« lire » un mot). La décomposition du mot en lettres doit être unique.

EXEMPLE 1.3. Soit l'alphabet

{0, 1, 00, 01, 10, 11}.

Un mot composé à partir des lettres de cet alphabet sera en général lu de différentes façons.

Par exemple, le mot

01001

peut être considéré comme

1) 0, 1, 0, 0, 0, 1,
2) 01, 00, 1,
3) 0, 10, 01,
4) 01, 0, 01,
5) 0, 1, 0, 01,
6) 0, 1, 00, 1,
7) 0, 10, 0, 1,
8) 01, 0, 0, 1.

Tout de même, si l'on donne la règle de formation des mots qui dit que les lettres d'un mot à deux symboles ne peuvent être que 0 et 1, et qu'un mot plus long commence par l'une des lettres 00, 01, 10, 11, les autres lettres ne pouvant être que 0 et 1, alors l'ambiguïté de lecture sera complètement supprimée. Le mot 01001 se lit alors d'une manière unique, comme 01, 0, 0, 1.

Remarquons qu'en acceptant la règle de formation de mots donnée ci-dessus, nous avons limité le nombre de mots aux seuls mots permis par cette règle.

Si les connecteurs sont figurés par exemple par les symboles {· →, →, →·}, alors l'éventualité de lectures différentes d'un mot est complètement éliminée. Ainsi, le mot envisagé plus haut s'écrira

· → 01 → 0 → 01 →·

de sorte qu'on ne saura le confondre avec aucun autre mot dans cet alphabet. Il est important de choisir les symboles des lettres et de connecteurs de manière à écarter toute ambiguïté lors de la décomposition d'une construction en ces symboles.

EXEMPLE 1.4. A l'aide des connecteurs de succession figurés, comme dans l'exemple 1.3, par les symboles {· →, →, →·} on peut, en plus de mots, composer certaines autres constructions. Soit l'alphabet des lettres

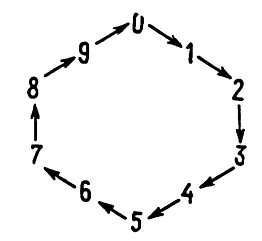
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

Si l'on admet que chaque connecteur peut être suivi de plusieurs connecteurs de prolongation, on peut obtenir des constructions

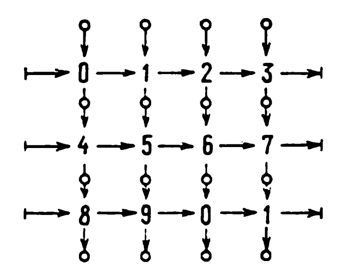
arborescentes, par exemple,

```
           3 → 4 → 7 →|
          ↗
↦ 1 → 2 → 5 → 6 →|
       ↘
        8 → 7 →|
         ↘
          9 → 0 → 0 →|
```

A l'aide des seuls connecteurs de prolongation on peut former des constructions assez curieuses, appelées anneaux:



Si l'on utilise un nouveau type de connecteurs figurés, par exemple, par les symboles ⫯ ⫱ ⫰, on pourra faire les constructions appelées matrices, ou tables:



En ajoutant un troisième type de connecteurs que nous interprétons par la disposition relative de lettres, comme nous l'avons déjà fait avant, nous pouvons construire les matrices de mots, par exemple,

```
  ⫯      ⫯     ⫯
↦ 291 →  5  → 12 →|
  ⫱      ⫱     ⫱
↦ 22  → 15  →  6 →|
  ⫰      ⫰     ⫰
```

ou les colonnes formées de lignes de mots, par exemple,

```
                    o
                    ↓
↦ 124 → 245 ──→ 45 →  56  → 7 ⇥
                    o
                    ↓
↦ 21  → 78  ──→ 0  →  2   ⇥
                    o
                    ↓
↦ 2   → 0   ──→ 111 → 1567 → 1 ⇥
                    ↓
                    o
```

Dans ce qui suit, il nous faudra effectuer le passage d'une numération donnée d'un nombre fini d'objets à n'importe quelle autre numération de ces objets. A cette fin considérons un système de fonctions $F_n(x, i)$; $i = 1, 2, \ldots, (n! - 1)$; $1 \leqslant x \leqslant n$, ($x$ entier) définies de la façon suivante. Soit la suite de nombres entiers

$$1, 2, \ldots, n.$$

Formons-en toutes les permutations possibles dont la quantité est $n!$ Numérotons-les d'une manière quelconque, avec la seule condition de donner le numéro un à la suite initiale. Composons tous les couples de permutations ayant la suite initiale pour premier élément. Chacun de ces couples peut être considéré comme une table des valeurs d'une fonction appliquant l'ensemble des numéros $1, 2, \ldots, n$ sur lui-même d'une façon biunivoque. Désignons la fonction qui est définie par la table composée de la première et de la $(i + 1)$-ème permutation par $F_n(x, i)$.

Envisageons à présent une classe de constructions (A, B).

Soit C un alphabet sans élément commun avec A ni avec B et dont le nombre des lettres est égal au genre maximal des connecteurs énumérés dans B. Soit, de plus, D un alphabet de trois lettres sans élément commun avec A, B, C. Pour fixer les idées, nous convenons que D se compose du chiffre I et de deux parenthèses (et).

Dans ce qui suit, nous dirons d'une lettre qui précède une autre lettre dans un alphabet qu'elle est *inférieure* à cette autre lettre.

Soit K une construction quelconque de classe (A, B). Il existe un procédé général qui fait correspondre à la construction K un mot bien déterminé P dans l'alphabet $A \cup B \cup C \cup D$. Décrivons ce procédé.

Si la construction K est vide, on lui fait correspondre un mot vide. Si K n'est pas vide, on procède comme suit.

E t a p e I. On trouve dans K, pour chaque lettre β_i ($i = 1, 2, \ldots$) de l'alphabet des connecteurs B, tous les connecteurs qui lui sont associés. S'il n'y en a qu'un seul, on le désigne par β_i, s'il y en a plusieurs, on les numérote arbitrairement et on les marque par les mots β_iI, β_iII, β_iIII, etc.

Ceci fait, tous les connecteurs figurant dans K se trouvent ordonnés, si l'on convient que le connecteur marqué par le mot β_i (par le mot β_iI . . . I, où I . . . I est le plus grand numéro arbitraire pour β_i) est immédiatement suivi du connecteur marqué par le mot β_{i+1} (ou par le mot β_{i+1}I).

Ensuite, en passant d'un connecteur à un autre dans l'ordre que l'on vient d'établir, on ordonne les branches de chaque connecteur de même qu'on l'a fait pour les connecteurs eux-mêmes, à ceci près qu'au lieu de l'alphabet B on se sert de l'alphabet C en posant qu'aux branches de genre i correspond la i-ème lettre de cet alphabet.

Tout en effectuant les opérations décrites on peut (chaque fois) tenir compte de tous les résultats possibles d'une numération arbitraire. A cette fin, chaque fois que l'on réalise une numération arbitraire, on reproduit la construction traitée (en $\varkappa! - 1$ exemplaires, si $\varkappa$ est le nombre des numéros arbitraires obtenus) d'une telle manière que les copies ne diffèrent de l'original que par des marques affectées lors de la dernière numération arbitraire. Pour le faire, on remplace dans chaque marque un numéro arbitraire I . . . I par un numéro arbitraire F_N (I . . . I, i) (où N est le nombre des numéros arbitraires et I . . . I le numéro de la copie).

Une fois la présente étape terminée, on a un certain nombre de constructions dans lesquelles tous les connecteurs et toutes leurs branches sont marqués par des numéros. Dans chaque construction, on trouve les connecteurs semi-extérieurs et l'on « coupe » celles de leurs branches qui lient les éléments intérieurs des constructions particulières liées spécialement, c'est-à-dire qu'on considère par la suite ces branches comme si elles étaient des connecteurs, et on considère ces connecteurs comme s'ils avaient moins de branches.

La description d'une branche de connecteur est un mot qui commence par une lettre dans B correspondant au type du connecteur. Cette lettre est suivie de plusieurs lettres I qui forment le numéro attribué au connecteur donné (lorsqu'il est unique, il n'est pas numéroté). Ensuite vient une lettre dans C correspondant au genre de la branche, et enfin plusieurs lettres I désignant le numéro de la branche dans le groupe des branches d'un même genre.

E t a p e II. Pour chacune des constructions obtenues on forme un mot P dans l'alphabet $A \cup B \cup C \cup D$.

Dans ce but, on choisit une lettre auxiliaire α n'appartenant pas à $A \cup B \cup C \cup D$ et on procède comme suit.

Règle de remplacement par un mot d'une construction à branches de connecteurs ordonnées

1°. On désigne par P le mot vide. On passe à 2°.

2°. Dans la construction K on trouve la branche (de connecteur) inférieure qui mènera à l'élément qu'elle lie. On passe à 3°.

3°. Si l'élément en question n'est pas une lettre, on le marque par le mot ($\alpha \ldots \alpha$) qui n'est pas encore utilisé et qui, en plus des parenthèses, contient seules les lettres α. Dans ce qui suit on convient d'appeler ce mot lettre. On passe à 4°.

4°. On écrit immédiatement le mot P après les noms de toutes les branches de connecteurs qui lient la lettre considérée, dans l'ordre que l'on leur a attribué, puis on met cette lettre. On obtient ainsi, à partir du mot P, un autre mot qui s'appelle dorénavant P. On passe à 5°.

5°. On trouve dans K la description de la branche inférieure qui ne figure pas dans P. D'après cette branche on trouve dans K l'élément qu'elle lie. On passe à 6°.

6°. On répète la séquence 3°-4°-5° jusqu à ce que toutes les branches soient épuisées. Puis on passe à 7°.

7°. On trouve dans le mot P la première parenthèse qui s'ouvre. Elle est suivie du mot $\alpha \ldots \alpha$; on trouve dans K la construction particulière qui lui correspond et on effectue sur celle-ci la séquence 1°-2°-3°-4° en remplaçant K par $\alpha \ldots \alpha$ et P par P'.

8°. Ceci fait, on inscrit P' dans P à la place du mot $\alpha \ldots \alpha$. On passe à 9°.

9°. On reprend la séquence 7°-8° autant de fois qu'il est nécessaire. L'étape se termine par l'obtention de P cherché.

Etape III. Après avoir effectué le procédé décrit pour toutes les constructions où nous avons marqué les connecteurs et leurs branches, nous obtenons une collection de mots. Pour description de la construction K on choisit celui des mots qui, si l'on considère l'alphabet E obtenu en écrivant successivement les alphabets A, B, C, D comme une liste des chiffres (0, 1, 2, . . .) d'un système de numération, s'avère être minimal. S'il y a plusieurs nombres minimaux (graphiquement ils sont identiques), on prend n'importe quel d'eux.

Par les opérations décrites on obtient une application des constructions sur les mots qui sera appelée *opération de linéarisation* (ou *linéarisation* tout court) de la construction K de classe (A, B). Il est aisé de voir qu'à toute construction K correspond un, et un seul, résultat de la linéarisation, dès que les alphabets C et D sont donnés.

Deux constructions de classe (A, B) sont dites *identiques* (et on peut les considérer comme des copies d'une même construction), si le résultat de leur linéarisation représente un même mot P (quels que soient les alphabets C et D).

Si la classe (A, B) de constructions est donnée et si P est le résultat de linéarisation d'une construction K de cette classe, alors il existe un procédé général qui permet de retrouver la construction K d après le mot P.

Le procédé de restitution d'une construction K d'après un mot P qui est sa description semble bien évident, il se réduit à un « monta-

ge » des connecteurs à partir de leurs branches, d'après les descriptions des branches contenues dans P.

L'opération de restitution d'une construction d'après un mot qui est sa description s'appelle *délinéarisation*.

En définissant la linéarisation et la délinéarisation nous avons utilisé une numération arbitraire d'éléments identiques. Néanmoins, le résultat de la linéarisation est univoque grâce à l'étape III. L'opération de délinéarisation est univoque en ce sens que toutes les constructions qu'elle fournit sont identiques.

1.1.3. Organisation générale d'un calculateur numérique. On distingue dans un calculateur numérique universel les trois parties principales (fig. 1.1) suivantes:

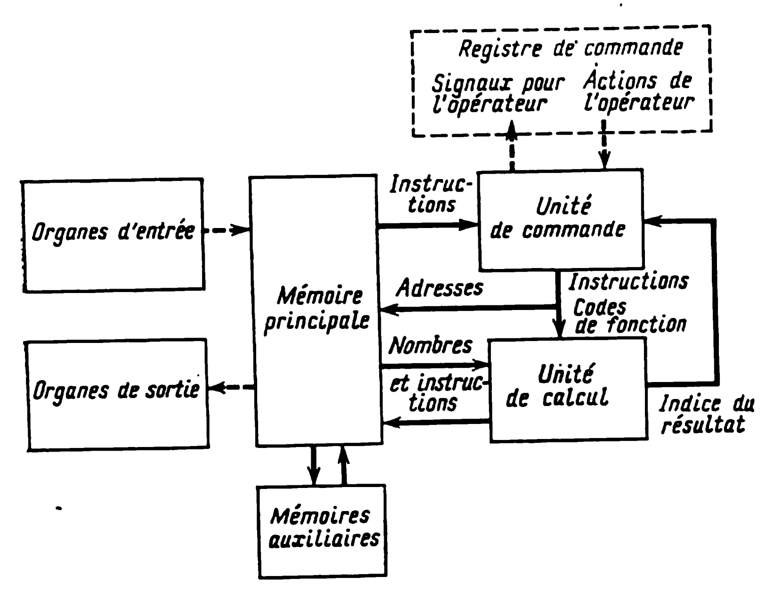


Fig. 1.1. Bloc-schéma simplifié d'un calculateur

— *l'unité de calcul* (ou de traitement) destinée à effectuer les opérations sur les mots (en particulier, sur les nombres);

— *les mémoires* où sont stockées les informations (en particulier, des nombres);

— *l'unité de commande* destinée à gérer le fonctionnement automatique de la machine.

Remarquons que les termes de *calculateur numérique* et d'unité de *calcul* que l'on applique à toutes les machines commandées par programmes, sont traditionnels. Ils datent de l'époque où les machines électroniques ne s'utilisaient que pour calculer; on continue à les

nommer ainsi, bien que les calculs représentent actuellement une partie insignifiante du domaine d'application de ces machines.

Un calculateur possède en général des mémoires de plusieurs types: une mémoire rapide de capacité moyenne qui constitue la *mémoire principale*, et des mémoires à accès lent et de grande capacité qu'on appelle *mémoires extérieures*, ou *mémoires auxiliaires* ou mémoires secondaires.

La mémoire rapide d'un calculateur est réservée aux mots qui servent directement à effectuer les opérations. Le reste de l'information est conservé dans la mémoire extérieure. L'échange d'informations entre la mémoire principale et les mémoires auxiliaires se fait au besoin.

L'enregistrement dans la mémoire rapide et l'extraction d'informations de celle-ci s'effectuent par des mots, quant à la mémoire auxiliaire, le rangement d'informations se fait par groupes de mots. Il est commode de considérer (et il en est ainsi pour la plupart des calculateurs actuels) que la mémoire rapide d'un calculateur digital se compose d'un certain nombre de *cellules*, ou *cases de rangement*, qui représentent une suite d'emplacements dont le nombre est égal au nombre d'*éléments binaires*, ou de *positions*, ou de *chiffres* du mot de machine.

A l'enregistrement d'un mot dans une cellule précède l'effacement du mot qui y est déjà contenu. Après l'enregistrement le contenu de la cellule reste invariable jusqu'à un nouvel enregistrement.

Dans la mémoire de masse les mots sont habituellement rangés par groupes, appelés *blocs*. On peut transférer dans la mémoire rapide ou bien un bloc tout entier, ou bien sa partie initiale composée de mots entiers.

En plus des trois parties principales signalées, un calculateur comprend des *organes d'entrée* des données, des *organes de sortie* des résultats et un *pupitre de commande*, ce dernier permettant l'intervention humaine dans le fonctionnement de la machine (voir fig. 1.1).

Pour la majorité des calculateurs existants, il est impossible d'introduire les données présentées arbitrairement. Le système universellement admis est celui de codage d'informations par des perforations sur un support mécanographique (*cartes* ou *rubans perforés*).

Les résultats peuvent être fournis soit sous forme de textes imprimés sur une large bande de papier, soit perforés sur les cartes ou sur le ruban.

Pour coder les données initiales sur les cartes (rubans) perforées et les entrer dans la machine on utilise les *machines* à *clavier*, ainsi que les *perforateurs d'entrée*, et pour imprimer les résultats sortis sous forme de cartes perforées, des *imprimantes* spéciales. Tous ces organes ne font pas partie du calculateur proprement dit, mais constituent son *équipement périphérique*.

Dans un proche avenir on pourra introduire dans la machine des textes typographiques ou dactylographiés, dans un avenir plus lointain, des textes écrits à la main ou même dictés. Il est possible d'introduire dans le calculateur des textes en provenance des canaux de communication, par exemple, d'un appareil télégraphique ou de l'organe de sortie d'un autre calculateur. La sortie de l'information peut s'effectuer sous forme de signaux électriques envoyés dans des canaux de communication, ou sous forme graphique, ou encore sur l'écran cathodique.

Les calculateurs modernes sont équipés de plusieurs unités de mémoire auxiliaire qui sont susceptibles de fonctionner simultanément. Il en est de même des organes d'entrée et de sortie.

En tant que mémoires auxiliaires on utilise les bandes et les tambours magnétiques qui échangent d'information avec la mémoire rapide par groupes de mots, ainsi que disques magnétiques qui, de plus, permettent l'accès direct à des mots enregistrés.

1.1.4. Principes de fonctionnement du calculateur. Le fonctionnement d'un calculateur est régi par les opérations qu'on appelle instructions.

Une *instruction* est une information représentée sous une forme compréhensible pour la machine et qui détermine le fonctionnement de celle-ci durant un certain temps.

Une suite d'instructions disposées d'une certaine façon dans la mémoire de la machine constitue le *programme*. Le programme est rédigé au cours d'une analyse préalable du problème à résoudre et est introduit dans la machine *) avec les données; ceci fait, la résolution s'effectue par la machine automatiquement.

Le principe de fonctionnement du calculateur selon lequel le traitement des données du problème et la restitution du résultat se font selon un programme composé à l'avance, s'appelle *principe de commande par programmes*. L'interprétation moderne de ce principe est donnée au § 1.4.

Dans la majorité des calculateurs existants, on utilise le *principe d'adressage* pour indiquer les *opérandes* (mots qui sont des données du problème ou des résultats des opérations) et, dans certains cas, les instructions. Le principe d'adressage consiste en ce que, pour désigner les mots gardés en mémoire, on utilise des mots standards appelés *adresses*. L'adresse d'un opérande représente en général le numéro (ou le nom) de la case de mémoire qui lui est affectée. On distingue dans une instruction une partie *fonction* et une partie *adresse*. La partie fonction, ou *code fonctionnel*, détermine ce que la machine doit effectuer, c'est-à-dire le type d'opération; la partie

*) Dans des machines spéciales, destinées à résoudre des problèmes bien fixes, le programme peut être mis en mémoire une fois pour toutes.

adresse contient une ou plusieurs adresses des opérandes (par exemple, les numéros des cases de mémoire où se trouvent les nombres objets de l'opération ou des cases désignées à en garder le résultat).

Il est à souligner que le terme « adresse » peut avoir deux significations: premièrement, c'est la désignation d'un opérande (pour la majorité des machines, du numéro de la case de mémoire qui le contient), et deuxièmement, la partie d'une instruction.

Ces derniers temps, on a vu apparaître des machines n'utilisant pas le principe d'adressage. Les instructions de telles machines comportent à la place des adresses des opérandes, les opérandes eux-mêmes. On verra si ces machines ont un avenir. Notons que, même pour les machines à adresses, certaines instructions exigent qu'on y inclue des opérandes.

Une fois le programme et les données du problème introduits dans le calculateur, son fonctionnement se ramène à la répétition d'un même *cycle de travail*. Deux schémas possibles de ce cycle sont donnés, à titre d'exemple, sur la figure 1.2, *a* et *b*, où les carrés et

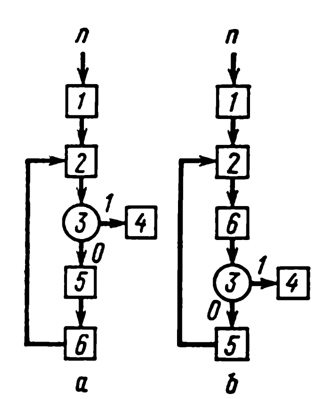


Fig. 1.2. Schémas possibles de cycles d'un calculateur

les cercles désignent des groupes d'opérations. П y désigne la mise en marche de la machine par l'opérateur; *1*, l'enregistrement de l'adresse de la première instruction du programme dans un registre spécial appelé *compteur ordinal* (CO); *2*, le transfert du mot dont l'adresse se trouve dans CO de la mémoire rapide dans le *registre d'instructions* (RI); *3*, la vérification si le contenu de RI est l'instruction d'arrêt et, dans l'affirmative, le passage dans le sens de la flèche marquée 1, dans le cas contraire, le passage dans le sens de la flèche marquée 0; *4*, l'arrêt du fonctionnement de la machine; *5*, l'exécution de l'instruction écrite dans RI; *6*, le « calcul » de l'adresse d'une nouvelle instruction et sa mise en CO.

Bien que de conception très variée, les calculateurs modernes fonctionnent tous suivant les schémas donnés sur la figure 1.2. Si l'on

fait une description détaillée du fonctionnement d'une machine concrète, le bloc *5* s'avère très embranché, certaines branches restant au sein du cycle, les autres menant à de nouveaux arrêts de la machine.

C'est là le *principe de fonctionnement cyclique*, car le résultat Y s'obtient à partir de la donnée initiale X sous la forme

$$Y = F(F(\ldots F(X))\ldots)$$

ou, plus exactement, sous la forme

$$Y = \Phi(\Phi(\ldots\Phi(P, X))\ldots),$$

où P est un prorgamme.

Lors du déroulement du programme, les instructions sont transférées une à une de la mémoire rapide dans l'unité de commande où elles sont exécutées dans un ordre donné à l'avance ou indiqué dans les instructions. Les mots lus dans la mémoire aux adresses indiquées dans une instruction (des instructions) sont fournis à l'unité de calcul qui effectue sur eux l'opération déterminée par le code de fonction; le résultat de l'opération peut être envoyé dans la case de mémoire dont le numéro est indiqué dans la partie adresse de l'instruction donnée ou d'une autre instruction.

L'unité de calcul envoie à l'unité de commande des signaux sur le caractère des résultats des opérations effectuées. Certains signaux font modifier l'ordre d'exécution des instructions du programme ou le caractère du fonctionnement de la machine. C'est ainsi qu'est réalisé le *principe d'action en retour*.

L'exécution de chaque instruction se déroule en plusieurs phases qui sont des opérations élémentaires dont les résultats sont retenus en mémoire et peuvent être utilisés lors des calculs ultérieurs. C'est le *principe d'exécution simultanée de plusieurs opérations élémentaires définies par une instruction*. Ce principe permet de réduire le temps nécessaire à la résolution d'un problème. Il explique la diversité de langages machine.

On a déjà dit que, dans la mémoire de calculateurs, l'information a la forme des mots qui se composent de lettres, et que le fonctionnement du calculateur représente une succession d'actions séparées. C'est le *principe de discontinuité de représentation et de traitement de l'information*.

Il est impossible de dire d'après l'aspect du mot machine s'il est une instruction ou un opérande. Cette distinction se révèle au cours d'exécution du programme: un mot de machine qui va au registre d'instructions sera considéré par la machine comme une instruction. Un mot envoyé dans l'unité de calcul ou qui en sort est considéré comme un opérande. Un programme doit être rédigé et disposé dans la mémoire d'une telle manière que seules les instructions arrivent dans le registre d'instructions. Un même mot est

susceptible de jouer, dans un calculateur, le rôle d'opérande et celui d'instruction. C'est l'essentiel du *principe d'unité d'instructions et d'opérandes.*

En plus de la fonction de traitement de l'information, un calculateur peut effectuer la vérification de certaines propositions (voir § 1.4) concernant l'information introduite ou traitée (voir § 2.3). On met en jeu la particularité des codes représentant les noms de certains objets (par exemple, de nombres) et ces objets eux-mêmes, autrement dit, entre la forme des mots machine et leur contenu. On voit là le *principe de formalité des langages machine.*

§ 1.2. Eléments de la théorie des langages formels

Des siècles durant, les sciences, et en particulier les mathématiques, avaient utilisé des langues naturelles, telles que le russe, le français, l'anglais, l'allemand, etc. Mais peu à peu il fut clair que l'utilisation de langues naturelles dans les sciences exactes conduit à des équivoques, voire à des contradictions. Voyons le problème de plus près. Dans un langage, il faut distinguer la structure des propositions (leur forme) et leur contenu (le sens). Dans les langues naturelles, ce n'est pas toujours qu'à la forme d'une proposition concrète correspond un contenu bien déterminé.

Notons ensuite que la science s'exprime dans le langage écrit. Or, les formes écrites des langues naturelles sont dépourvues de certaines possibilités des formes parlées; il s'agit d'intonation et de gestes. Par exemple, les trois phrases suivantes, qui ne diffèrent que par l'intonation (par l'accent d'insistance) ont les sens différents: 1. « Je *pars* demain matin », 2. « Je pars *demain* matin », 3. « Je pars demain *matin* ». Naturellement, on pourrait bien introduire certaines règles pour exprimer de telles nuances d'une langue naturelle dans les textes écrits, mais ceci au prix d'énormes complications et sans certitude de n'avoir rien oublié.

Enfin, citons le paradoxe connu de Richard, sous la forme que lui a donnée Berry (v. [19],) : « Le plus petit nombre naturel qui ne peut être nommé en français avec moins de vingt mots. » Par cette phrase qui contient 17 mots on a nommé en français un objet qu'il est impossible de nommer en français avec moins de 20 mots. C'est un paradoxe (une contradiction).

Donc, les langues naturelles sont contradictoires, ambigües, floues. De plus, pour comprendre certaines propositions exprimées dans une langue naturelle, il faut se placer dans le contexte de la situation.

Le moyen de pallier ces inconvénients des langues naturelles est né au cours de leur développement. Au sein de ces langues, à base de leur vocabulaire et leur grammaire, se forment des langages dont les phrases sont univoques et ont un sens qui ne dépend pas du con-

texte. Le sens de chaque phrase d'un tel langage est déterminé par sa seule *forme*. Les différentes notations de nombres servent d'exemples de tels langages *formalisés*. Lorsqu'on s'est rendu compte des avantages de ces langages, on a conçu l'idée de construction de langages formels artificiels, libres de restrictions propres à des langues naturelles et orientés vers les applications déterminées.

Si la phrase citée en tant que paradoxe de Richard, était exprimée dans une autre langue, il n'y aurait rien de paradoxal. Il en résulte que la possibilité même du paradoxe de Richard sera éliminée si l'on adopte des langages formels dans lesquels il est impossible de parler d'eux-mêmes. Pour décrire un langage formel, on aura besoin d'un autre langage. Le premier de ces deux langages s'appelle *langage-objet*, le deuxième, *métalangage*.

Quelquefois, on est amené à se servir de plusieurs métalangages, ainsi que de langages qui sont des métalangages par rapport à d'autres métalangages, c'est-à-dire des métamétalangages. Dans les descriptions ci-dessous, le langage de départ est l'une des langues naturelles, notamment le français. En parlant d'un langage, il faut faire une nette distinction entre le langage-objet et des métalangages pour éviter des confusions.

Une construction achevée dans une langue naturelle est ordinairement appelée *phrase*. Nous garderons ce terme pour des langages formels. Un métalangage doit avoir les possibilités de décrire aussi bien la structure que le sens des phrases du langage-objet. On distingue souvent dans un métalangage deux langages dont l'un est destiné à la description de la structure des phrases du langage-objet et l'autre, à la description de leur sens. Le premier est alors appelé *métalangage syntaxique*, le deuxième, *métalangage sémantique*. Le système de règles qui déterminent la structure des phrases du langage-objet est appelé sa *syntaxe*; la correspondance entre les phrases du langage-objet et leur sens, sa *sémantique*.

1.2.1. Syntaxe d'un langage formel. Dans les langues naturelles on trouve des constructions bien compliquées. Ainsi, les mots d'une langue sont composés de lettres à l'aide de connecteurs du premier type; à partir des mots et à l'aide de connecteurs du deuxième type, on compose des groupes, à la manière dont on forme les mots à partir de lettres. A leur tour, les groupes sont associés, à l'aide de connecteurs du troisième type, en des chaînes qui représentent des phrases.

EXEMPLE 1.5. Dans la phrase française « L'homme qui marchait dans la rue était de bonne humeur: il agitait les bras et souriait », on distingue, au niveau des mots, les connecteurs du premier type interprétés par la disposition relative des lettres; les mots sont enchaînés à l'aide de connecteurs du deuxième type représentés par des espaces entre les mots; les groupes de mots sont enchaînés à l'aide

de connecteurs du troisième type figurés par les symboles « , » et « : ». Le connecteur de début du troisième type est traduit par la majuscule au début de la phrase, le connecteur de fin du troisième type est donné par le symbole « . », et les connecteurs de prolongation du troisième type sont de plusieurs espèces. Les connecteurs de début et de fin du deuxième type sont interprétés par la disposition relative de mots.

Notons que, si l'on considérait les symboles désignant les connecteurs comme des lettres, on aurait pu traiter la phrase donnée comme un seul mot (on devrait alors considérer les espaces eux aussi comme lettres).

Nous accepterons des langages formels dont les phrases sont d'une structure plus compliquée que les constructions des langues naturelles. D'autre part, nous considérerons des langages dont les phrases sont des mots.

En décrivant la structure des phrases d'un langage-objet on indique habituellement les alphabets des lettres et des connecteurs, et l'on formule les règles de composition des phrases (dans le cas simple on omet l'alphabet des connecteurs). Pour fixer les idées, nous convenons que chaque règle nomme *) une *opération* que l'on peut appliquer à la construction des phrases du langage-objet et indique les opérandes. Ainsi, nous sous-entendons par syntaxe d'un langage formel une liste d'opérations dont on se donne les domaines de définition. De plus, nous supposons que la syntaxe comprend la formulation d'une *condition* qui est remplie pour des constructions achevées du langage-objet (même si elles sont utilisées pour des constructions ultérieures) et n'est pas remplie pour des constructions qui ne sont pas des phrases du langage-objet.

1.2.1.1. Codes. Les langages formels qu'on appelle *codes* sont largement répandus. Un métalangage d'un tel langage formel se compose de deux langages (plus exactement, sous-langages); le premier représente l'ensemble des phrases qui ont un sens et se rapportent au domaine qui nous intéresse, le deuxième contient la description de deux règles qu'on appelle règle de codage et règle de décodage. La syntaxe du langage-objet représente la règle de codage qui, appliquée à une phrase du premier sous-langage, fournit une phrase du langage-objet. La condition dont on a parlé plus haut est omise dans ce cas puisqu'elle dit : « une construction est le résultat de l'application de la règle de codage à une phrase du premier sous-langage du métalangage », et chaque résultat de codage lui satisfait. La règle de décodage définit une opération inverse de celle de codage (cette dernière doit donc obligatoirement admettre une

*) Nous disons « nomme » et non « décrit », puisque la description peut être donnée dans l'un des métalangages décrivant le langage concerné.

opération inverse). Disons en anticipant que la règle de décodage détermine la sémantique du code (voir 1.2.2). Ajoutons encore que, pour qu'un code soit un langage formel, son métalangage doit l'être lui aussi. Le lecteur trouvera quelques exemples de codes qui sont des langages formels aux points 2.2.3 à 2.2.8.

1.2.1.2. Notation normale de Backus. De nos jours on utilise largement pour la construction de métalangages syntaxiques, une méthode appelée notation normale de Backus. D'après cette méthode, un métalangage syntaxique est défini par un ensemble de règles de grammaire qu'on appelle métaformules linguistiques (de Backus). Pour les construire on emploie deux métasymboles universels: ::= et | dont le premier se lit comme *est par définition*, le deuxième comme *ou* et donc permet de séparer les différentes possibilités. Les autres métasymboles sont choisis au gré de la personne qui construit le langage-objet et le métalangage. Les métasymboles sont des mots ou des groupes de mots quelconques d'une langue naturelle mis entre chevrons. Il est d'usage d'employer comme métasymboles les noms des catégories grammaticales du langage-objet. Ces métasymboles seront appelés composés. En plus de métasymboles, on se sert dans les formules de Backus des symboles du langage-objet qui sont faciles à reconnaître car ils diffèrent de ::= et | et n'ont pas de chevrons.

Dans la partie de gauche d'une formule de Backus doit être écrit un métasymbole composé. Il est suivi du symbole ::=, puis vient la partie de droite. Elle peut être vide, ou bien représenter une suite finie de métasymboles composés et (ou) de symboles du langage-objet, ou enfin une suite finie de constructions de ce genre séparées par les symboles |. Par définition, deux formules de Backus ayant la même partie de gauche et les parties de droite différentes ont la même signification qu'une seule formule que l'on obtient en écrivant après la partie de droite de l'une des formules le symbole |, puis la partie de droite de l'autre. De même, une formule de Backus qui contient à la partie de droite un symbole | est équivalente à deux formules dont les parties de gauche sont identiques à celle de la formule donnée et dont les parties de droite sont respectivement identiques aux segments de la partie de droite de la formule donnée séparés par le symbole mentionné |.

Comme le début et la fin d'une formule de Backus ne sont pas marqués, il est bon de commencer l'écriture avec un retrait et laisser une marge à la fin de la ligne. En écrivant une formule sur plusieurs lignes on ne met aucun signe de séparation supplémentaire. L'interprétation d'une formule de Backus ne présente pas de difficultés si on la lit dans la langue naturelle (on ne lit pas les chevrons; les métasymboles ::= et | se lisent de la façon indiquée plus haut; une ligne vide dans la partie de droite d'une formule se lit « vide »).

Exemple 1.6. Un langage formel pour la représentation des nombres entiers pairs peut être décrit comme suit dans la notation de Backus:

⟨chiffre non nul⟩ : : = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 (1)
⟨chiffre⟩ : : = 0 | ⟨chiffre non nul⟩ (2)
⟨chiffre pair⟩ : : = 0 | 2 | 4 | 6 | 8 (3)
⟨début⟩ : : = ⟨chiffre non nul⟩ | ⟨début⟩ ⟨chiffre⟩ (4)
⟨nombre pair⟩ : : = ⟨chiffre pair⟩ | ⟨début⟩ ⟨chiffre pair⟩ (5)

Le métasymbole ⟨nombre pair⟩ désigne une phrase du langage-objet. Le métalangage en question définit 4 et 103 comme ⟨nombres pairs⟩, c'est-à-dire comme phrases du langage-objet. En effet, en vertu de (3), 4 est l'une des valeurs du métasymbole ⟨chiffre pair⟩ et en vertu de (5), 4 est l'une des valeurs du métasymbole ⟨nombre pair⟩. Exactement de même, d'après (1) 1 est un ⟨chiffre non nul⟩;, d'après (4), 1 est un ⟨début⟩, et, comme 0 est un ⟨chiffre⟩ selon (2), 10 est aussi un ⟨début⟩ en vertu de la deuxième partie de (4). Enfin, (3) implique que 8 est un ⟨chiffre pair⟩, et d'après la deuxième partie de (5) 108 est un ⟨nombre pair⟩.

La notation de Backus est bonne pour la description d'un langage-objet dont les phrases sont des mots. Ces mots s'obtiennent en réunissant, dans l'ordre de succession, les symboles du langage-objet et les valeurs de métasymboles composés écrits dans les parties de droite des formules de Backus.

Remarquons qu'on ne peut pas décrire à l'aide de la notation de Backus tout langage formel, même si ses phrases sont des mots. Si par exemple on avait dans 1.6 au lieu de ⟨nombre pair⟩ la phrase ⟨nombre pair double⟩, il serait impossible d'écrire la formule correspondante de Backus.

Il est à noter que les métasymboles composés dans les formules de Backus sont toujours employés comme un tout et doivent donc être considérés en tant que lettres du métalangage. La condition à laquelle doit satisfaire chaque phrase du langage-objet est qu'une phrase « ne contienne pas de métasymboles et soit obtenue comme valeur de l'une des suites de la partie de droite d'une formule dont le premier membre est le métasymbole désignant une phrase ». On omet de formuler cette condition dans le métalangage en convenant qu'elle est sous-entendue chaque fois qu'on utilise la notation de Backus.

1.2.1.3. Métalangages inductifs. Une généralisation des formules de Backus sont les *métaformules universelles* (voir [28]). On peut réaliser cette généralisation de la manière suivante. Supposons qu'un langage formel *L* soit décrit à l'aide d'un métalangage *M* écrit dans la notation de Backus. Transformons les formules de Backus du

métalangage M d'une façon équivalente (si c'est possible) en faisant en sorte qu'aucune formule ne contienne le métasymbole | dans sa partie de droite. Désignons le nouveau système de formules de Backus par M'. A présent, la partie de droite de chaque formule est une suite de symboles représentant le résultat de fusion en un seul mot des mots du langage-objet et des mots désignés par les symboles composés. Notons S_i $(\alpha_1, \alpha_2, \ldots, \alpha_i)$ l'opération de réunion de i mots $\alpha_1, \alpha_2, \ldots, \alpha_i$ en un mot $\alpha_1\alpha_2 \ldots \alpha_i$. Alors $S_1(\alpha_1) = \alpha_1$, c'est-à-dire que S_1 est tout simplement l'opération identique. En remplaçant dans les formules de Backus du métalangage M' les parties de droite (qui ont la forme $\alpha_1\alpha_2 \ldots \alpha_i$) par les notations correspondantes S_i $(\alpha_1, \alpha_2, \ldots, \alpha_i)$, nous arrivons à un nouveau métalangage (désignons-le M'') dont chaque formule est de la forme (nous mettons « ; » à la fin)

$$\alpha ::= S_i(\alpha_1, \alpha_2, \ldots, \alpha_i);$$

Le nouveau métalangage représente un code du métalangage M'. Il décrit le même langage L que M'. Il faut que les nouveaux symboles que nous avons introduits dans le métalangage ne soient pas ceux du langage-objet. Dans la notation de Backus, cette restriction ne concernait que deux métasymboles principaux et les chevrons. Les arguments des parties de droite de nouvelles formules sont ou bien des constructions du langage-objet, ou bien des métasymboles composés; dans les parties de gauche, il n'y a que des symboles composés. L'ensemble des constructions du langage-objet figurant dans les parties de droite des formules sera appelé *base* du langage-objet, et ces constructions elles-mêmes, *morphèmes*. Dans notre cas les morphèmes sont des mots.

La généralisation mentionnée de la notation de Backus consiste en ceci.

— Nous admettons en tant que morphèmes du langage-objet des constructions arbitraires d'une classe (A, B).

— En tant qu'opérations indiquées dans les parties de droite des métaformules, nous admettons les opérations qui s'effectuent sur les constructions de la classe (A, B) et dont les résultats appartiennent à la même classe.

— Nous supprimons la condition qui exige que tous les métasymboles, sauf deux symboles principaux, soient construits à l'aide de chevrons de la manière décrite plus haut (en n'interdisant tout de même pas l'utilisation de symboles composés) ; nous indiquerons chaque fois la liste (l'alphabet) des métasymboles, et nous choisirons un métasymbole (métasymbole distingué) pour désigner la notion « phrase du langage-objet ». Les symboles utilisés pour nommer les opérations, de même que les parenthèses, la virgule et le point-virgule seront des métasymboles.

— Nous convenons qu'en plus des connecteurs utilisés dans le langage-objet, le métalangage possède les connecteurs de succession (de début, de prolongation et de fin) qui sont différents de tous les connecteurs du langage-objet (parmi lesquels il peut y avoir également des connecteurs de succession).

Remarque. Si quelques-uns des symboles qu'on a choisis comme métasymboles figurent dans l'alphabet du langage-objet, il faut les remplacer dans le métalangage par d'autres symboles, pour éviter que l'alphabet du langage-objet contienne des métasymboles.

Les formules obtenues par la généralisation décrite plus haut s'appellent *métaformules universelles*, et un métalangage qui se compose d'un nombre fini de telles formules est dit *générateur* par rapport au langage-objet (il l'engendre à partir de morphèmes). Pour qu'une construction obtenue à l'aide d'une métaformule universelle soit vraiment celle du langage-objet il faut qu'« elle soit obtenue à l'aide d'une formule dont la patrie de gauche est le métasymbole distingué ».

Le métalangage M'' dont on a dit qu'il est un code du métalangage M' représente un cas spécial de métalangage générateur récursif (inductif). C'est exactement en ce sens-là que nous dirons que le métalangage M (donné dans la notation de Backus) représente un cas spécial de métalangage inductif.

Exemple 1.7. Un métalangage inductif, équivalent au métalangage de l'exemple 1.6, peut être composé des formules universelles suivantes:

$a ::= S_1(1)$; $a ::= S_1(2)$; $a ::= S_1(3)$; $a ::= S_1(4)$;
$a ::= S_1(5)$; $a ::= S_1(6)$; $a ::= S_1(7)$; $a ::= S_1(8)$;
$a ::= S_1(9)$; $c ::= S_1(0)$; $c ::= S_1(a)$; $b ::= S_1(0)$;
$b ::= S_1(2)$; $b ::= S_1(4)$; $b ::= S_1(6)$; $b ::= S_1(8)$;
$n ::= S_1(a)$ $n ::= S_2(n, c)$; ⟨nombre pair⟩ $::= S_1(b)$;
⟨nombre pair⟩ $::= S_2(n, b)$;

Pour comprendre ce métalangage, il faut savoir qu'il est inductif, que le langage-objet a l'alphabet des lettres 0123456789 et l'alphabet des connecteurs (qui est omis ici) se composant de trois connecteurs de succession ordinaires, que l'alphabet des métasymboles a la forme

$$abcn \text{ ⟨nombre pair⟩} ::= S_1 S_2 (\ ),;$$

et que le métasymbole distingué est ⟨nombre pair⟩.

Le métalangage de l'exemple 1.6 est évidemment bien plus commode pour la description du langage des nombres pairs, mais

ceci est dû au fait qu'il représente un cas spécial qui convient à la situation. Il serait impossible de l'appliquer à la description d'un langage-objet dont les phrases sont des textes disposés en colonnes de lignes qui, à leur tour, se composent de nombres pairs. Or, il est aisé de décrire un tel langage-objet à l'aide de métaformules universelles. Pour le faire, ajoutons à l'alphabet des connecteurs du langage-objet trois connecteurs de succession des nombres pairs (nous n'emploierons aucun symbole pour désigner les connecteurs de début et de fin et nous interprétons par un espace blanc le connecteur de prolongation) et trois connecteurs de succession des lignes (les connecteurs de début et de fin ne seront pas désignés et le connecteur de prolongation sera interprété par la disposition des lignes, chaque ligne étant placée au-dessous de la précédente); en outre, introduisons les métasymboles S_1^* et S_2^* pour décrire les opérations de fusion de lignes, et W_1 et W_2 pour désigner les opérations de fusion de colonnes; enfin, introduisons les métasymboles d et ⟨texte⟩. Considérons comme distingué non pas le métasymbole ⟨nombre pair⟩ mais le métasymbole ⟨texte⟩. Ajoutons aux métaformules vues plus haut les quatre métaformules suivantes:

$$d ::= S_1^*(\langle\text{nombe pair}\rangle);\ d ::= S_2\ (d,\ \langle\text{nombre pair}\rangle);$$
$$\langle\text{texte}\rangle ::= W_1(d);\ \langle\text{texte}\rangle ::= W_2(\langle\text{texte}\rangle,\ d).$$

Remarquons que notre mode de représentation des connecteurs n'est pas obligatoire. On peut les figurer par n'importe quels symboles différant des lettres.

1.2.2. Sémantique d'un langage formel. A l'heure actuelle, il n'existe pas de définition mathématique de la notion de sémantique d'un langage (même formel). On peut en donner évidemment une pseudo-définition: on appelle sémantique d'un langage L une application qui fait correspondre à toute phrase de ce langage un élément déterminé d'un certain ensemble de notions. Pourtant, deux choses sont ici inconnues: ce qu'on entend par notion et comment se donner la correspondance en question. Sans savoir définir la sémantique dans le sens général, on peut pourtant formuler, dans des cas spéciaux, les conditions dans lesquelles un langage possède une sémantique. La définition de la notion de sémantique d'un langage se réduira donc forcément à une énumération des cas spéciaux. Malheureusement, quelque grand que soit le nombre des cas spéciaux énumérés, il reste toujours des doutes quant à la généralité de la définition. La notion de sémantique des langages formels sera détaillée au § 5.5.5, ici nous ne considérons que quelques éléments dont nous aurons besoin dans les chapitres qui suivent.

Lorsque, pour un langage formel L, on donne une règle exacte et univoque de transformation de ses phrases en celles d'un autre langage possédant une sémantique, alors la sémantique de notre

langage L se trouve bien définie. Une sémantique est dite formelle si la règle en question est un algorithme, au sens que nous attribuons à ce terme au § 1.5. En particulier, le rôle de ce second langage formel peut être tenu par un langage d'une langue naturelle. Les codes du p. 1.2.1.1 représentent les exemples simples des langages ayant une sémantique ainsi définie.

Si le langage-objet sert à la description des transformations à faire subir à certaines données (qui sont des phrases du second langage formel) pour obtenir les résultats cherchés, alors une sémantique formelle du langage-objet est donnée dès que sera connue une règle exacte et univoque (un alogrithme au sens du § 1.5) indiquant ce qu'il faut faire pour obtenir, à partir d'une phrase du langage-objet et d'une donnée, le résultat cherché.

Comme on l'a déjà dit, un langage formel est donné si sont données sa syntaxe et sa sémantique. Tout de même, dans certains problèmes, seule la syntaxe du langage formel est essentielle. Par exemple, une telle situation a lieu pour un langage formel dont les phrases sont des données initiales des transformations mentionnées plus haut. Dans certains cas, on a à traiter des constructions produites de la même façon que des langages formels, mais qui n'ont pas de sémantique. Il est parfois commode de les envisager comme des langages. Ceci vu, introduisons la notion de langage formel à sémantique vide. Nous sommes amenés à dire que tout ensemble de constructions défini à l'aide d'un métalangage syntaxique représente un langage. Pour fixer les idées, nous convenons de dire qu'un langage formel est un langage-objet défini à l'aide d'un métalangage inductif. Si, de plus, une sémantique du langage formel est définie, nous dirons que c'est un langage à sémantique.

Dans le cas contraire, nous dirons que le langage formel a une sémantique vide.

Pour conclure ce point, arrêtons-nous sur la sémantique d'un métalangage syntaxique inductif. Chaque phrase d'un tel langage représente une règle exacte et univoque (un alogrithme au sens du § 1.5) permettant d'obtenir les constructions d'une certaine classe d'après les collections de constructions d'autres classes. Le nom de la classe est le métasymbole dans la partie de gauche de la métaformule (de la phrase). Il n'est pas rare qu'une classe de constructions complètement définie non pas par une seule, mais par plusieurs métaformules ayant les parties de gauche identiques. Le langage-objet est assimilé à la classe de constructions nommée par le métasymbole distingué. Il est évident que par sa structure toute classe représente un langage formel. Souvent, tous ces langages formels, sauf, éventuellement, le langage-objet, sont des langages à sémantique vide. Nous pouvons conclure que, d'après la manière de définir la sémantique, un métalangage inductif se rapporte au second cas spécial signalé plus haut.

3—0779

§ 1.3. Systèmes de numération

On appelle *numération* un ensemble de procédés de nomination et de notation des nombres.

Les symboles conventionnels que l'on utilise pour représenter les nombres sont appelés *chiffres*.

Dans des systèmes de numération usuels, les nombres sont représentés comme une suite de chiffres. On distingue deux types de systèmes de numération: *positionnels* et *non positionnels*, suivant que la valeur d'un chiffre varie ou non en fonction de sa position dans la suite. En tant qu'exemple d'un système non positionnel, signalons le système *romain*. Nous n'étudions ici que les systèmes positionnels.

1.3.1. Systèmes de numération positionnels. Soit p un nombre entier supérieur à l'unité que nous appelons *base* d'un système de numération. Choisissons p symboles deux à deux distincts qui seront les *chiffres* de notre système. De plus, choisissons p nombres entiers successifs dont zéro. Cette suite de nombres sera appelée *support* du système de numération. *Etablissons une correspondance biunivoque entre les chiffres et les nombres du support.*

Nous ne considérons ici que les systèmes de numération dont les supports sont soit *non négatifs* (se composent de zéro et de nombres positifs), soit *symétriques* par rapport à zéro. Les chiffres du système de numération qui correspondent aux nombres du support symétriques par rapport à zéro sont dits *symétriques*.

Exemple 1.8. Citons à titre d'exemple d'un système de numération positionnel le système décimal d'usage universel. Le nombre p est ici égal à dix, le support est non négatif et se compose de dix nombres entiers successifs de zéro à neuf. En tant que chiffres décimaux on utilise les chiffres arabes 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Etant donné un système de numération de base p, un nombre s'écrit sous la forme d'une suite de chiffres qui est partagée par une virgule en deux sous-suites. Si chacun des symboles $a_n, a_{n-1}, \ldots$ $\ldots, a_1, a_0, a_{-1}, \ldots, a_{-m}$ désigne un chiffre, alors un nombre se note sous la forme:

$$a_n a_{n-1} \ldots a_1 a_0, a_{-1} \ldots a_{-m}. \tag{1.1}$$

Les positions qu'occupent les chiffres dans la suite (1.1) sont considérées comme numérotées: les positions à gauche de la virgule sont numérotées de droite à gauche, dans l'ordre de leur succession, par les nombres zéro, un, deux, etc.; les positions à droite de la virgule sont numérotées de gauche à droite, dans l'ordre de leur succession,

par les nombres moins un, moins deux, etc. Ces positions numérotées sont appelées *rangs*.

On associe une valeur bien déterminée à tout chiffre de la suite (1.1). Un chiffre occupant la position nulle a pour valeur le nombre du support qui lui correspond. Tout chiffre d'un nombre a la valeur p fois plus grande que la valeur qu'il aurait à la position immédiatement inférieure et une valeur p fois plus petite que celle qu'il aurait à la position immédiatement supérieure.

Une suite de chiffres désigne un nombre égal à la somme des valeurs de ses chiffres.

Ceci dit, la suite (1.1) signifie

$$a_n a_{n-1} \ldots a_1 a_0, a_{-1} \ldots a_{-m} =$$
$$= a_n p^n + a_{n-1} p^{n-1} + \ldots + a_1 p + a_0 + a_{-1} p^{-1} + \ldots + a_{-m} p^{-m}. \quad (1.2)$$

Dans ce qui suit, s'il n'y a pas de confusion à craindre, nous dirons « nombre » au lieu de « notation d'un nombre ».

La suite des chiffres d'un nombre placés à gauche de la virgule représente la *partie entière* du nombre. Si le support d'un système de numération contient des nombres positifs et négatifs, alors il existe dans ce système des fractions propres ayant une partie entière (voir, par exemple, page 38). La suite que l'on obtient en remplaçant par zéro la partie entière d'un nombre s'appelle *partie fractionnaire*.

EXEMPLE 1.9. Exemples de la notation décimale : a) la base du système (dix) se note 10; b) $8401{,}302 = 8\cdot 10^3 + 4\cdot 10^2 + 0\cdot 10^1 + 1\cdot 10^0 + 3\cdot 10^{-1} + 0\cdot 10^{-2} + 2\cdot 10^{-3}$. Pour écrire dans le système décimal des nombres négatifs, on utilise, en plus des chiffres, le signe « — » (*moins*) que l'on met devant le nombre (i.e. à gauche de la suite de chiffres) ; parfois on met le signe « + » (*plus*) devant un nombre positif.

L'addition, la soustraction, la multiplication et la division des nombres représentés dans un système de base p s'effectuent bien simplement à l'aide des tables d'addition, de soustraction et de multiplication, d'une manière analogue à ce qu'on fait dans le système décimal. La multiplication d'un nombre par la base p se réduit, en vertu de la formule (1.2), au déplacement de la virgule d'une position vers la droite, et la division par p, au déplacement de la virgule d'une position vers la gauche.

Dans les calculateurs et dans les programmes établis pour les calculateurs on utilise outre le système décimal d'autres systèmes de numération. Nous allons étudier quelques systèmes des plus usités.

Système binaire à support non négatif. Les chiffres de ce système sont 0 et 1. Le nombre « deux » (la base du système) est noté comme

10. En écrivant les nombres négatifs on fait précéder la suite de chiffres du signe moins.

Les tables d'addition (table 1.1), de soustraction (table 1.2)

Table 1.1

Table d'addition binaire

0+0=0
0+1=1
1+0=1
1+1=10

Table 1.2

Table de soustraction binaire

0−0=0
1−0=1
1−1=0
10−1=1

Table 1.3

Table de multiplication binaire

0×0=0
0×1=0
1×0=0
1×1=1

et de multiplication (table 1.3) binaires sont très simples.

Avec ces tables, l'addition, la soustraction, la multiplication et la division des nombres binaires s'effectuent d'après les mêmes règles que nous avons l'habitude d'utiliser en additionnant, soustrayant, multipliant et divisant les nombres décimaux.

EXEMPLE 1.10.

a) addition

```
 1 100 111,011
+   10 011,111
--------------
 1 111 011,010
```

b) soustraction

```
 10 110,1101
-10 001,1111
------------
    100,1110
```

c) multiplication

```
   1100111,1101
×        11,011
---------------
   1100 1111101
  11001 111101
1100111 1101
11001111 101
---------------
101011110,0101111
```

d) division

```
11011101101 | 1001
1001        |--------
----        | 11000101
 1001
 1001
 ----
   1011
   1001
   ----
    1001
    1001
    ----
    0000
```

Système octal à support non négatif. Ce système utilise huit chiffres différents 0, 1, 2, 3, 4, 5, 6, 7. La base du système (huit) est noteé 10. En écrivant les nombres négatifs on met le signe moins devant la suite de chiffres.

Les tables d'addition (table 1.4) et de multiplication (table 1.5)

Table 1.4

Table d'addition octale

+	0	1	2	3	4	5	6	7	10
0	0	1	2	3	4	5	6	7	10
1	1	2	3	4	5	6	7	10	11
2	2	3	4	5	6	7	10	11	12
3	3	4	5	6	7	10	11	12	13
4	4	5	6	7	10	11	12	13	14
5	5	6	7	10	11	12	13	14	15
6	6	7	10	11	12	13	14	15	16
7	7	10	11	12	13	14	15	16	17
10	10	11	12	13	14	15	16	17	20

Table 1.5

Table de multiplication octale

×	0	1	2	3	4	5	6	7	10
0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	10
2	0	2	4	6	10	12	14	16	20
3	0	3	6	11	14	17	22	25	30
4	0	4	10	14	20	24	30	34	40
5	0	5	12	17	24	31	36	43	50
6	0	6	14	22	30	36	44	52	60
7	0	7	16	25	34	43	52	61	70
10	0	10	20	30	40	50	60	70	100

octales sont ici « à deux entrées ».

EXEMPLE 1.11. A.

a) addition

$$\begin{array}{r} 327,71102 \\ +\ 35,67735 \\ \hline 365,61037 \end{array}$$

b) soustraction

$$\begin{array}{r} 11076,01 \\ -\ 705,62 \\ \hline 10170,16 \end{array}$$

EXEMPLE 1.11. B.

a) multiplication

```
   173.261
    16,35
  --------
  1150565
  562023
1344046
173261
----------
3366.56615
```

b) division

```
336656,615 | 1635
1635       | 173,261
----
15315
14513
-----
  6026
  5327
  ----
   4776
   3472
   -----
   13041
   12656
   -----
    1635
    1635
    ----
    0000
```

Système hexadécimal à support non négatif. Le système utilise seize chiffres différents *) 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *a*, *b*, *c*, *d*, *e*, *f*. La base du système (seize) s'écrit 10. En écrivant les nombres négatifs, on met le signe moins devant la suite de chiffres.

Les tables d'addition et de multiplication dans le système hexadécimal sont beaucoup plus volumineuses que les tables décimales. Il est aisé de les construire par analogie avec les tables 1.4 et 1.5.

Système ternaire à support symétrique. Dans ce système on utilise *trois* chiffres différents $\bar{1}$, 0, 1. La base du système (trois) s'écrit comme 10. On n'a pas besoin du signe (moins) pour distinguer les nombres positifs et négatifs. Le chiffre de rang élevé d'un nombre négatif est toujours $\bar{1}$, celui d'un nombre positif, 1. Dans ce système il peut exister des fractions propres ayant la partie entière non nulle. Par exemple, le nombre 2/3 s'écrit comme 1, $\bar{1}$.

L'addition et la soustraction des nombres s'effectuent à l'aide de la table 1.6.

Table 1.6

Table d'addition ternaire

+	$\bar{1}$	0	1
$\bar{1}$	$\bar{1}1$	$\bar{1}$	0
0	$\bar{1}$	0	1
1	0	1	$1\bar{1}$

*) Pour désigner les chiffres supérieurs à 9, on utilise aussi bien d'autres symboles, par exemple *A*, *B*, *C*, *D*, *E*, *F* ou $\bar{0}$, $\bar{1}$, $\bar{2}$, $\bar{3}$, $\bar{4}$, $\bar{5}$, etc.

EXEMPLE 1.12. A.

a) addition

$$
\begin{array}{r}
1100\bar{1},10\bar{1}\bar{1} \\
+\quad 1\bar{1}00,01\bar{1}0 \\
\hline
1\bar{1}\bar{1}\bar{1}0\bar{1},101\bar{1}
\end{array}
$$

b) soustraction

$$
\begin{array}{r}
1\bar{1}\bar{1}\bar{1}0\bar{1}1,01\bar{1} \\
-\quad 1\bar{1}000,1\bar{1}0 \\
\hline
01100\bar{1}1,0\bar{1}\bar{1}
\end{array}
$$

La multiplication d'un nombre par l'unité négative se réduit au remplacement de tous les chiffres de ce nombre par les chiffres symétriques. Ceci permet de diminuer le volume de la table de multiplication (table 1.7) en y mettant seuls les produits des nombres non négatifs du support. Les produits des autres nombres du support s'en obtiennent à l'aide de la règle usuelle des signes pour la multiplication.

Table 1.7

Table de multiplication ternaire (cas de support symétrique)

×	0	1
0	0	0
1	0	1

Avec les tables 1.6 et 1.7, la multiplication des nombres s'effectue d'après les mêmes règles que dans le système décimal.

EXEMPLE 1.12. B. Multiplication

$$
\begin{array}{r}
110\bar{1},\bar{1}01 \\
\times\quad 10,\bar{1}1 \\
\hline
1\bar{1}01101\bar{1} \\
110\bar{1}\bar{1}01 \\
110\bar{1}\bar{1}01 \\
\hline
11011,\bar{1}111\bar{1}
\end{array}
$$

L'algorithme de la division utilise les tables 1.6 et 1.7 et diffère un peu de celui du système décimal.

Cette différence consiste en ceci. Lorsque le dividende intermédiaire (qu'on obtient en complétant le reste par un chiffre abaissé du dividende) contient autant de chiffres que le diviseur, alors, que le dividende intermédiaire soit supérieur ou non au diviseur, on écrit dans le quotient le chiffre 1 si les premiers chiffres du dividende et du diviseur sont identiques et l'on écrit le chiffre $\bar{1}$ dans le cas contraire. Si ensuite le nouveau reste contient autant de chiffres

que le diviseur, on répète l'opération indiquée en écrivant le nouveau chiffre du quotient non pas à droite du précédent, mais au-dessous de lui. Après avoir terminé le procédé (qu'on arrête ou bien en obtenant le reste nul, ou bien artificiellement, tout comme pour la division décimale), le nombre à la première ligne du quotient est à additionner avec le nombre de la deuxième ligne, en considérant les espaces blancs de la deuxième ligne comme zéros.

EXEMPLE 1.12. C. Division

$$\begin{array}{l|l}
101\bar{1}0\bar{1}1\bar{1}\bar{1} & \;1\bar{1}\bar{1} \\ \cline{2-2}
\underline{1\bar{1}\bar{1}} & 100\bar{1}001 \quad + \\
1\bar{1}\bar{1} & \underline{1\quad\bar{1}\qquad\quad} \\
\underline{1\bar{1}\bar{1}\quad\;} & 1\bar{1}0\bar{1}1001 \\
000\bar{1}0\bar{1} & \\
\quad\underline{\bar{1}11} & \\
\quad\bar{1}11 & \\
\quad\underline{\bar{1}11\quad\;} & \\
\quad 00011\bar{1} & \\
\qquad\underline{1\bar{1}\bar{1}} & \\
\qquad 000 &
\end{array}$$

Système symétrique de base neuf. Dans ce système de numération on utilise neuf chiffres différents $\bar{4}$, $\bar{3}$, $\bar{2}$, $\bar{1}$, 0, 1, 2, 3, 4. La base du système (neuf) est notée comme 10. Pour distinguer les nombres positifs et négatifs on n'a pas besoin d'un signe (moins). Un nombre est négatif ou positif selon que son chiffre de rang élevé est représenté par un nombre négatif ou positif du support.

L'addition et la soustraction des nombres s'effectuent à l'aide de la table 1.8.

Table 1.8

Table d'addition de base neuf (cas de support symétrique)

+	$\bar{4}$	$\bar{3}$	$\bar{2}$	$\bar{1}$	0	1	2	3	4
$\bar{4}$	$\bar{1}1$	$\bar{1}2$	$\bar{1}3$	$\bar{1}4$	$\bar{4}$	$\bar{3}$	$\bar{2}$	$\bar{1}$	0
$\bar{3}$	$\bar{1}2$	$\bar{1}3$	$\bar{1}4$	$\bar{4}$	$\bar{3}$	$\bar{2}$	$\bar{1}$	0	1
$\bar{2}$	$\bar{1}3$	$\bar{1}4$	$\bar{4}$	$\bar{3}$	$\bar{2}$	$\bar{1}$	0	1	2
$\bar{1}$	$\bar{1}4$	$\bar{4}$	$\bar{3}$	$\bar{2}$	$\bar{1}$	0	1	2	3
0	$\bar{4}$	$\bar{3}$	$\bar{2}$	$\bar{1}$	0	1	2	3	4
1	$\bar{3}$	$\bar{2}$	$\bar{1}$	0	1	2	3	4	$1\bar{4}$
2	$\bar{2}$	$\bar{1}$	0	1	2	3	4	$1\bar{4}$	$1\bar{3}$
3	$\bar{1}$	0	1	2	3	4	$1\bar{4}$	$1\bar{3}$	$1\bar{2}$
4	0	1	2	3	4	$1\bar{4}$	$1\bar{3}$	$1\bar{2}$	$1\bar{1}$

Exemple 1.13.

a) addition

$$\begin{array}{r} 4\bar{4}3\bar{2}0\bar{1}1 \\ +\ \bar{3}20\bar{4}14 \\ \hline 33\bar{4}\bar{2}\bar{4}1\bar{4} \end{array}$$

b) soustraction

$$\begin{array}{r} 33\bar{4}\bar{2}\bar{4}1\bar{4} \\ -\ 4\bar{4}3\bar{2}0\bar{1}1 \\ \hline 0\bar{3}20\bar{4}14 \end{array}$$

Systèmes de numération codés binaires. Soit p la base d'un système de numération positionnel. A tout son chiffre faisons correspondre d'une manière biunivoque, un nombre entier binaire. Déterminons le nombre k de chiffres du plus grand de ces derniers et faisons en sorte que tous les nombres binaires choisis aient le même nombre de chiffres, en ajoutant à gauche autant de zéros qu'il le faut. A chaque chiffre il correspond maintenant un nombre binaire de k chiffres qu'on appelle son *code binaire*. Tout nombre de base p peut être codé en remplaçant chacun de ses chiffres par son code binaire. Ce système de notation des nombres s'appelle *système codé binaire*. Il est évident que le plus petit nombre possible de chiffres dans les codes binaires s'obtient si l'on choisit k de façon à satisfaire à l'inégalité

$$2^{k-1} < p \leqslant 2^k.$$

Il est aisé de voir qu'il reste $2^k - p$ nombres binaires de k chiffres qui ne sont pas utilisés comme codes de chiffres en base p. On appelle ces nombres combinaisons « interdites » (de zéros et d'unités).

Exemple 1.14. Considérons le système décimal codé binaire dans lequel chaque chiffre décimal est codé par son équivalent binaire de quatre chiffres. Le nombre de combinaisons interdites est alors $2^4 - 10 = 6$ (les voici : 1010, 1011, 1100, 1101, 1110, 1111). Les groupes de quatre chiffres binaires qui représentent les codes des chiffres décimaux sont appelés *tétrades*. Le nombre décimal 893,2 sera noté dans notre système comme 1000 1001 0011, 0010.

Pour certaines machines électroniques, on adopte le système décimal codé binaire de notation des nombres. Les opérations arithmétiques sur les codes binaires des nombres décimaux s'effectuent alors d'après les règles spéciales. Dans les machines qui travaillent sur les codes binaires purs le système décimal codé binaire est utilisé comme intermédiaire, pour introduire les nombres dans la machine.

Notations combinées de nombres. Pour réduire la notation de nombres en base p ou de n'importe quelles suites de chiffres codés on utilise parfois une notation dite *combinée*. On groupe les chiffres et on remplace chaque groupe par un seul chiffre du système de base p^k, où k est le nombre de chiffres dans un groupe. On peut même utiliser des systèmes de notation de bases différentes (les groupes pouvant avoir les nombres différents de chiffres). Pour décoder une

notation combinée, i.e. déterminer le nombre en base p qu'elle représente, on indique pour chaque position le système de notation utilisé.

EXEMPLE 1.15. Les codes binaires

10111101

01100001

peuvent être écrits en notation combinée comme

2 75

1 41,

où le chiffre de poids fort est de base quatre, les autres étant de base huit.

La notation combinée s'emploie généralement pour figurer les instructions et nombres sur des formulaires. Afin d'éviter le décodage, les dispositifs à clavier qui servent à introduire les informations dans la machine sont spécialement adaptés à la notation combinée utilisée.

1.3.2. Passage d'un système de numération à un autre.

1[er] cas. Il est facile de passer de la notation d'un nombre en base p à sa notation en base q (ou vice versa) lorsque $p = q^k$ (k un entier positif) et les deux systèmes ont des supports non négatifs ou symétriques.

Dans ce cas, le passage du système de base p à celui de base q se fait « par chiffres », en remplaçant chaque chiffre en base p par le nombre de k chiffres écrit dans le système de base q. Le passage du système de base q à celui de base p s'effectue de la façon suivante. En partant de la virgule vers la gauche et vers la droite on sépare dans la notation en base q des groupes de k chiffres. Si les groupes extrêmes s'avèrent incomplets, on y ajoute respectivement à gauche ou à droite autant de zéros qu'il faut pour que le groupe contienne k chiffres. Puis on remplace chaque groupe de chiffres en base q par un seul chiffre en base p égal au nombre exprimé par ce groupe.

EXEMPLE 1.16. Systèmes de numération à supports non négatifs:

a) un nombre noté 273,54 dans le système octal s'écrit en base binaire ($8 = 2^3$) comme suit:

$$273{,}54 = (010)\ (111)\ (011),\ (101)\ (100) = 10111011{,}1011;$$

b) la conversion du binaire 11011,0011 en octal ($8 = 2^3$) se fait comme suit:

$$11011{,}0011 = 11\ (011),\ (001)\ 1 = (011)\ (011),\ (001)\ (100) = 33{,}14;$$

c) un nombre hexadécimal $a5,\ b1e$ s'écrit en binaire ($16 = 2^4$) comme suit:

$$a5,\ b1e = (1010)\,(0101),\ (1011)\,(0001)\,(1110) = 10100101,\ 10110001111.$$

EXEMPLE 1.17. Systèmes de numération à supports symétriques:

a) un nombre noté $2\bar{4}1\bar{2},01\bar{3}$ en base neuf se transforme en base trois ($9 = 3^2$) comme suit:

$$2\bar{4}1\bar{2},01\bar{3} = (1\bar{1})\,(\bar{1}\bar{1})\,(01)\,(\bar{1}1),\quad (00)\,(01)\,(\bar{1}0) = 1\bar{1}\bar{1}\bar{1}01\bar{1}1,\quad 0001\bar{1};$$

b) passons de la notation ternaire $110\bar{1}\bar{1}01,110\bar{1}\bar{1}$ à la notation en base neuf ($9 = 3^2$):

$$110\bar{1}\bar{1}01,\ 110\bar{1}\bar{1} = 1\,(10)\,(\bar{1}\bar{1})\,(01),\ (11)\,(0\bar{1})\,\bar{1} =$$
$$= (01)\,(10)\,(\bar{1}\bar{1})\,(01),\ (11)\,(0\bar{1})\,(\bar{1}0) = 1\bar{3}41,\ 4\bar{1}\bar{3}.$$

2e cas. Le passage d'un système de numération à un autre ne présente pas de difficultés non plus si les bases de deux systèmes sont identiques mais les supports sont différents. Ce passage consiste en ce que chaque chiffre du système de numération donné qui n'appartient pas au nouveau système est considéré comme somme (ou différence) de deux chiffres du nouveau système. Dans la notation du nombre, on remplace un tel chiffre de l'ancien système par le premier des deux chiffres choisis du nouveau système, et l'on écrit le second chiffre au-dessous du premier. Au-dessous des chiffres appartenant aux deux systèmes de numération on écrit des zéros. On obtient deux nombres écrits dans le nouveau système, on les additionne (ou soustrait) selon les règles de ce nouveau système.

EXEMPLE 1.18. Ecrire les nombres $1\bar{1}01\bar{1},\ 0\bar{1}$ et $\bar{1}\bar{1}0011,\ 1\bar{1}1$ donnés dans le système ternaire de support symétrique dans le système ternaire de support non négatif (0, 1, 2). Pour effectuer ce passage, nous considérons le chiffre $\bar{1}$ comme la différence 0—1 des chiffres du nouveau système.

Pour le nombre $1\bar{1}01\bar{1},0\bar{1}$ on a:

$$-\begin{array}{r} 10010,00 \\ 01001,01 \\ \hline 2001,22 \end{array}.$$

Par conséquent,

$$1\bar{1}01\bar{1},0\bar{1} = 2001,22.$$

Pour le nombre $\bar{1}\bar{1}0011,1\bar{1}1$ on a:

$$-\begin{array}{r} 000011,101 \\ 110000,010. \end{array}$$

Puisque cette fois le premier terme de la différence est inférieur au deuxième et le support du nouveau système de numération ne contient pas de nombres négatifs, nous soustrayons le premier nombre au deuxième en mettant devant le résultat le signe moins. Nous obtenons

$$\bar{1}\bar{1}0011{,}1\bar{1}1 = -102211{,}202.$$

Exemple 1.19. Pour passer de la notation 7803,12 en base neuf à supports non négatifs à la notation en base neuf à support symétrique nous posons

$$7 = 4 + 3; \quad 8 = 4 + 4.$$

Nous obtenons

$$\begin{array}{r} 4403{,}12 \\ +\ 3400{,}00 \\ \hline 1\bar{1}\bar{1}03{,}12 \end{array}$$

Ainsi,

$$7803{,}12 = 1\bar{1}\bar{1}03{,}12$$

3e cas. Le passage d'un système de base p à un système de base q, lorsque les supports des deux systèmes sont non négatifs et $p \neq q^k$, se fait séparément pour les parties entière et fractionnaire du nombre (pour les détails, voir, par exemple, [37]).

Transformation de la partie entière. On écrit le nombre q en base p. On divise par q, en base p la partie entière du nombre traité ; le reste représente le dernier chiffre de la notation cherchée en base q. Ensuite, on divise par q le quotient obtenu ; le nouveau reste fournit l'avant-dernier chiffre de la notation en base q, etc. On poursuit le procédé jusqu'à ce qu'on obtienne un quotient inférieur à q qui constituera le premier chiffre de la notation cherchée.

Exemple 1.20. Ecrire en octal le nombre décimal 191 :

$$\begin{array}{r|l} 191 & 8 \\ \cline{2-2} \underline{16} & 23 \;|\; \underline{8} \\ 31 & \underline{16} \quad 2 \\ \underline{24} & \;\;7 \\ 7 & \end{array}$$

Le nombre décimal 191 se note en octal 277.

Conversion de la partie fractionnaire. On écrit le nombre q dans le système de base p. On multiplie la fraction, en base p, par q. La partie entière du produit donne le premier chiffre de la notation en base q de la fraction. On multiplie la partie fractionnaire du produit toujours par q. La partie entière du nouveau produit fournit le chiffre suivant de la notation cherchée de la fraction. On termine l'opération ayant obtenu soit un produit entier, soit un nombre de chiffres demandé de la notation en base q.

EXEMPLE 1.21. Trouver la notation octale du nombre décimal 0,6875 :

0	6875
	8
5	5000
	8
4	0000

On obtient la notation de base huit 0,54.

En combinant les règles de passage signalées on peut aisément passer d'un système de numération positionnel à n'importe quel autre système positionnel.

Parfois on effectue le passage d'un système quelconque au système décimal à l'aide de la formule (1.2) (puisqu'on n'a pas l'habitude d'effectuer la division dans des systèmes autres que décimal). On écrit dans la représentation décimale les chiffres de base p, le nombre p et les exposants contenus au second membre de la formule, après quoi on calcule la valeur de l'expression obtenue selon les règles du système décimal.

1.3.3. Représentation des nombres dans les calculateurs. On représente les nombres dans la machine soit *en virgule fixe*, soit *en virgule flottante*.

Représentation en virgule fixe. En virgule fixe, chaque nombre est représenté par la suite de ses chiffres. Dans les calculateurs utilisant cette forme, la place de la virgule est fixée une fois pour toutes. Elle est donc la même pour tous les nombres. Ceci permet d'omettre la virgule dans la représentation du nombre. Si le système de numération utilisé est de support non négatif, alors en plus des chiffres du nombre, il faut figurer son signe. On prévoit donc les positions spéciales pour le signe.

Lorsqu'on fixe la virgule après une position déterminée, l'erreur absolue de représentation ne dépasse pas l'unité de rang le moins élevé, ce qu'on exprime en disant que l'erreur absolue maximale est fixe.

Les calculateurs utilisant la forme en virgule fixe, c'est-à-dire dont les organes de calcul sont spécialement construits pour les opérations sur des nombres mis sous cette forme, sont appelés *calculateurs à virgule fixe*. Certaines opérations sur les nombres représentés en virgule fixe conduisent à des résultats qu'il est impossible de mettre sous cette forme dans la machine. Par exemple, lorsqu'on multiplie deux fractions impropres, il peut arriver que le nombre de chiffres de la partie entière du produit dépasse le nombre des positions réservées dans la machine. Une telle situation s'appelle *débordement de capacité*. Si l'on n'admet que les fractions propres,

c'est-à-dire si l'on fixe la virgule à gauche du chiffre significatif de rang le plus élevé, alors la multiplication ne peut conduire au débordement de capacité. C'est pour cette raison que, dans la plupart des machines à virgule fixe, on place la virgule exactement de cette façon.

Si le nombre des chiffres de la partie fractionnaire d'un nombre dépasse la capacité de la machine il faudra conserver les poids forts, le dernier chiffre retenu étant peut-être arrondi. Il y a évidemment perte de précision, un résultat pouvant à la limite être obtenu égal à 0 pour des opérandes différents de 0 mais suffisamment petits. Un nombre qui n'est pas effectivement nul mais apparaît comme tel dans la machine s'appelle zéro de la machine.

Représentation en virgule flottante. Dans ce cas on suppose que chaque nombre est mis sous la forme

$$N = mp^n, \tag{1.3}$$

où m est un nombre, appelé *mantisse*, dans lequel la virgule est fixée après une position bien déterminée, n est un nombre entier appelé *exposant*, p est la base du système de numération utilisé dans la machine. Un nombre en virgule flottante sera représenté par un couple de nombres m, n, i.e. par sa mantisse, et par son exposant. Si le système de numération utilisé est de support non négatif alors la mantisse et l'exposant seront représentés avec leurs signes, et l'on prévoit des positions spéciales pour les signes.

Il est d'usage de fixer la virgule des mantisses devant le chiffre de rang le plus élevé, c'est-à-dire que l'on représente les mantisses par des fractions propres. Ceci est dû à l'avantage que présente la multiplication de telles mantisses (leur produit sera toujours une fraction propre).

Si, dans la notation (1.3), le premier chiffre après la virgule n'est pas nul, le nombre N est dit *normalisé* (et *non normalisé* dans le cas contraire). Les nombres normalisés sont donc ceux qui vérifient la relation

$$\frac{1}{p} \leqslant |m| < 1.$$

EXEMPLE 1.22. Le nombre décimal 25,2 peut avoir, en virgule flottante, n'importe quelle des représentations suivantes : $0{,}252 \cdot 10^2$, $0{,}0252 \cdot 10^3$, $0{,}00252 \cdot 10^4$, etc. Seule la notation $0{,}252 \cdot 10^2$ est normalisée, les autres ne le sont pas.

Lorsque le nombre des positions réservées aux chiffres de la mantisse est μ, alors l'erreur relative de représentation en virgule flottante des nombres normalisés ne dépasse pas la quantité $p^{1-\mu}$, où p est la base du système de numération utilisé. Pour les nombres non normalisés, l'erreur relative peut atteindre de très grandes valeurs.

Les calculateurs pour lesquels on adopte la représentation en virgule flottante, c'est-à-dire dont les organes de calcul sont spécialement construits pour les opérations sur des nombres mis sous cette forme, sont appelés *calculateurs à virgule flottante.*

Dans les calculateurs à virgule flottante, un *débordement de capacité* est aussi possible. Il s'agit cette fois du débordement de capacité réservée à l'exposant. Il se peut bien que, dans les machines à virgule flottante, apparaissent des *zéros de la machine*, c'est-à-dire des nombres normalisés non nuls dont l'exposant est inférieur au plus petit exposant représentable dans la machine; on écrit le zéro de la machine sous la forme d'une mantisse « nulle » et d'un exposant « nul ».

Lorsque le nombre de chiffres d'une mantisse dépasse celui qui est réservé pour la mantisse dans la machine, on conserve les poids forts, en rejetant les poids faibles, avec l'arrondissement éventuel du dernier chiffre gardé.

Intervalle de représentation des nombres. L'ensemble des nombres que l'on peut représenter est une caractéristique essentielle de la machine. Pour les machines à virgule fixe qui utilisent un système de numération à support non négatif ou symétrique, cet ensemble contient le zéro, est symétrique par rapport au zéro (c'est-à-dire qu'il contient à la fois le nombre N et le nombre $-N$) et peut être mis sous forme d'une progression arithmétique croissante ayant pour raison le plus petit nombre positif représentable dans la machine. C'est pourquoi, pour décrire l'ensemble des nombres représentables en virgule fixe il suffit d'indiquer le plus petit et le plus grand nombres positifs qu'on peut représenter dans la machine ou, comme on dit encore, l'intervalle de représentation des nombres (plus précisément, de leurs valeurs absolues).

L'intervalle de représentation des nombres dans une case de mémoire d'une machine à virgule fixe, qui utilise le système de numération binaire, peut être estimé à l'aide des inégalités

$$2^{-\nu} \leqslant |N| \leqslant 2^{n} - 2^{-\nu},$$

où N est le nombre, n, la quantité d'éléments binaires réservés à la partie entière du nombre, ν, la quantité d'éléments binaires réservés à la partie fractionnaire (sans compter les signes).

Pour les calculateurs à virgule flottante qui utilisent un système de numération de support non négatif ou symétrique, l'ensemble des nombres représentables dans une case de mémoire s'obtient en indiquant l'intervalle des mantisses (qui sont des nombres représentés en virgule fixe) et la plus grande valeur possible de l'exposant (les exposants sont des entiers, l'ensemble des exposants contient le zéro et est symétrique par rapport au zéro). Pour caractériser une machine à virgule flottante, au lieu de donner une description complète de l'ensemble des nombres représentables, on indique

habituellement l'intervalle de représentation (plus précisément, l'intervalle des valeurs absolues), ou même l'intervalle des nombres normalisés.

Pour des machines à virgules flottantes qui utilisent le système de numération binaire l'intervalle des nombres représentables normalisés (virgule de la mantisse à gauche) est estimé par les inégalités

$$2^{-2^{\rho}} \leqslant |N| \leqslant (1-2^{-\mu})\, 2^{2^{\rho}-1},$$

où N est, comme plus haut, le nombre, μ, la quantité d'éléments binaires réservés aux chiffres de la mantisse, ρ, la quantité d'éléments binaires réservés aux chiffres de l'exposant.

L'intervalle de représentation des nombres des calculateurs à virgule flottante est beaucoup plus large que celui des calculateurs à virgule fixe, pour une même longueur du mot machine.

De gros calculateurs modernes sont capables généralement de traiter aussi bien en virgule fixe qu'en virgule flottante, c'est-à-dire permettent les deux formes de représentation des nombres. Dans de telles machines, pour les nombres en virgule fixe on adopte la position de la virgule dite « à droite », c'est-à-dire (immédiatement) après le chiffre de rang le moins élevé, ce qui correspond aux nombres entiers. Dans ces machines tous les calculs se font en virgule flottante, sauf les paramètres qu'on traite en virgule fixe.

§ 1.4. Eléments de logique mathématique

La *logique mathématique* est une partie de la logique générale qui est développée spécialement pour les besoins des mathématiques, avec une large application de méthodes mathématiques. Longtemps la logique mathématique n'avait été qu'une science étudiant les démonstrations mathématiques. Avec l'apparition de calculateurs numériques commandés par programmes, elle acquit une valeur appliquée. La possibilité d'appliquer la logique mathématique à la résolution de problèmes à l'aide de calculateurs électroniques est liée à l'analogie qui existe entre le calcul des propositions et des prédicats d'une part, et la théorie des fonctions binaires de variables binaires d'autre part. Il s'avère que l'on peut calculer les valeurs des fonctions logiques de même que celles des fonctions mathématiques.

1.4.1. Notions de connecteurs logiques, de valeur logique et de proposition. Le calcul des propositions est la partie la plus simple de la logique mathématique. On y considère des propositions composées qui s'obtiennent à partir des propositions dites élémentaires soit en les faisant précéder de la particule « non », soit en les reliant à l'aide des conjonctions « et », « ou », « équivaut » et « si . . .,

alors . . . ». La particule « non » et les conjonctions mentionnées sont appelées *connecteurs logiques.*

Le calcul des propositions ne s'intéresse pas au sens de ces dernières, mais estime leur valeur de vérité. On dit d'une proposition vraie que sa valeur logique est le **vrai**, et d'une proposition fausse, que sa valeur logique est le **faux**. De même qu'en mathématiques on considère les nombres qui sont susceptibles d'être valeurs de grandeurs (mathématiques), dans la logique mathématique on considère qu'une variable logique peut prendre deux valeurs : **vrai** et **faux**. Les mots **vrai** et **faux** sont imprimés en caractères demi-gras pour souligner que ce ne sont pas des mots ordinaires français, mais une sorte de « nombres » de la logique mathématique.

Toute proposition dont il est possible de dire qu'elle est vraie ou qu'elle est fausse et considérée sans tenir compte de sa structure interne ni de son sens, s'appelle *proposition élémentaire.*

Les propositions « Neuf est un nombre impair », « La neige est rouge », « Moscou est la capitale de l'U.R.S.S. » sont des propositions élémentaires.

Par contre, les phrases de type « coupez l'allumage en sortant », « quel âge avez-vous ? », etc., ne sont pas des propositions et la logique mathématique ne les examine pas. La logique mathématique ne détermine pas la valeur logique d'une proposition élémentaire. Une proposition élémentaire est considérée comme telle dès qu'elle possède l'unique valeur logique que l'on suppose connue.

Dans une langue naturelle, le sens des mots « et », « ou »,« équivalence », « si . . ., alors . . . », « non » est parfois flou, et certains d'entre eux peuvent s'employer aux sens différents. Par exemple, dans la langue française, « ou » peut être exclusif (comme dans la phrase « choisis, lui ou moi ») ou non exclusif (comme dans la phrase « j'étais réveillé par un bruit ou par une lumière », où la possibilité « par les deux à la fois » n'est pas exclue). En logique mathématique, le sens de la particule « non » et des conjonctions mentionnées se précise de la manière suivante. Pour abréger l'écriture, désignons les mots « non », « et », « ou », « équivalence », « si . . ., alors . . . » respectivement par les symboles — (barre surmontant une lettre ou un mot), $\wedge$, $\vee$, $\sim$, $\rightarrow$. On admet que, lorsque A est une proposition, alors $\bar{A}$ est aussi une proposition, la valeur logique, ou valeur de vérité, de cette dernière étant opposée à celle de A.

Ainsi, le sens de la particule « non » dans la logique mathématique est qu'elle modifie les valeurs logiques selon la table 1.A.

Table 1.A

$\overline{\textbf{faux}} = \textbf{vrai}$

$\overline{\textbf{vrai}} = \textbf{faux}$

D'une manière analogue, le sens des autres connecteurs logiques est clair à partir des tables 1.B, 1.C, 1.D, et 1.E.

Table 1.B

faux $\wedge$ faux = faux
faux $\wedge$ vrai = faux
vrai $\wedge$ faux = faux
vrai $\wedge$ vrai = vrai

Table 1.C

faux $\vee$ faux = faux
faux $\vee$ vrai = vrai
vrai $\vee$ faux = vrai
vrai $\vee$ vrai = vrai

Table 1.D

faux $\rightarrow$ faux = vrai
faux $\rightarrow$ vrai = vrai
vrai $\rightarrow$ faux = faux
vrai $\rightarrow$ vrai = vrai

Table 1.E

faux $\sim$ faux = vrai
faux $\sim$ vrai = faux
vrai $\sim$ faux = faux
vrai $\sim$ vrai = vrai

On voit que le sens qu'on attribue en logique mathématique aux connecteurs logiques n'est pas complètement adéquat à leur signification dans la langue naturelle (dans le langage courant).

Formulons une définition formelle d'une *proposition* :

1) une proposition élémentaire est une proposition ;

2) si A et B sont des propositions, alors $\bar{A}$ (se lit : *non* A), $A \wedge B$ (se lit : A *et* B), $A \vee B$ (se lit : A *ou* B), $A \sim B$ (se lit : A *équivaut* B) et $A \rightarrow B$ (se lit : *si* A, *alors* B) sont encore des propositions ; on détermine les valeurs logiques des propositions composées à l'aide des tables ci-dessus.

Lorsque les propositions composantes A et B sont elles-mêmes composées (i.e. s'écrivent en utilisant des lettres et des connecteurs logiques), on les met entre parenthèses dans les notations dont il est question au point 2) de la définition.

Ainsi, en écrivant une proposition composée on emploie des lettres (propositions élémentaires), des connecteurs et des parenthèses (qui déterminent la structure des propositions composées).

Soulignons que le calcul des propositions ne fournit aucun moyen de détermination de la valeur de vérité des propositions élémentaires, mais donne des règles bien nettes permettant de connaître celle des propositions composées d'après la vérité des propositions composantes (et non d'après leur sens). Les propositions composées $A \wedge B$, $A \vee B$, $A \sim B$, $A \rightarrow B$ sont respectivement appelées *conjonction*, *disjonction*, *équivalence* et *implication* des propositions composantes A et B.

1.4.2. Fonctions logiques et opérations fondamentales de l'algèbre de Boole. Une variable x qui prend les valeurs 0 et 1 s'appelle variable *binaire*.

Une fonction de n variables binaires

$$F(x_1, x_2, \ldots, x_n)$$

qui ne peut prendre que deux valeurs 0 et 1, ou même l'une de ces valeurs, s'appelle *fonction logique.*

Il n'existe que quatre fonctions logiques différentes d'une variable binaire, à savoir celles données dans la table 1.9.

Table 1.9

Fonctions logiques d'une variable binaire

x \ $f(x)$	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$
0	0	0	1	1
1	0	1	0	1

Deux de ces fonctions sont des constantes : $f_1(x) = 0$; $f_4(x) = 1$.

La fonction $f_2(x)$ est habituellement désignée par le symbole $\bar{\bar{x}}$. La table 1.9 montre que cette fonction satisfait à la relation

$$\bar{\bar{x}} = x. \qquad (1.4)$$

Il est convenu de désigner la fonction $f_3(x)$ par le symbole $\bar{x}$. Le signe « ‾ » peut être considéré non seulement comme signe de la fonction, mais aussi comme celui de l'opération qui donne d'après la valeur de x (égale à 0 ou à 1) celle de la fonction $\bar{x}$. Cette opération s'appelle *inversion* et on peut la décrire par la table 1.10 qui résulte de la table 1.9.

Table 1.10

Inversion

$$\bar{0} = 1$$
$$\bar{1} = 0$$

On voit de la table 1.10 que

$$\overline{(\bar{x})} = x. \qquad (1.5)$$

Les formules (1.4) et (1.5) permettent d'interpréter le signe « $=$ » comme celui d'une double inversion. Dans ce qui suit on pose

$$\bar{\bar{x}} = \overline{(\bar{x})}. \qquad (1.6)$$

Il existe seize fonctions logiques différentes de deux variables binaires (table 1.11).

Table 1.11

Fonctions logiques de deux variables binaires

x \ F	y \ F	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table 1.12

Multiplication logique

$$0 \wedge 0 = 0$$
$$0 \wedge 1 = 0$$
$$1 \wedge 0 = 0$$
$$1 \wedge 1 = 1$$

Table 1.13

Addition logique

$$0 \vee 0 = 0$$
$$0 \vee 1 = 1$$
$$1 \vee 0 = 1$$
$$1 \vee 1 = 1$$

Table 1.14

Equivalence

$$0 \sim 0 = 1$$
$$0 \sim 1 = 0$$
$$1 \sim 0 = 0$$
$$1 \sim 1 = 1$$

Table 1.15

Implication

$$0 \rightarrow 0 = 1$$
$$0 \rightarrow 1 = 1$$
$$1 \rightarrow 0 = 0$$
$$1 \rightarrow 1 = 1$$

Deux de ces fonctions sont des constantes: $F_1(x, y) = 0$, $F_{16}(x, y) = 1$.

Les fonctions $F_2(x, y)$, $F_8(x, y)$, $F_{10}(x, y)$, $F_{14}(x, y)$ sont habituellement désignées par les symboles respectifs $x \wedge y$, $x \vee y$, $x \sim y$, $x \rightarrow y$. On peut considérer les symboles $\wedge$, $\vee$, $\sim$, $\rightarrow$ comme ceux des opérations binaires qui permettent d'obtenir, à partir de deux nombres x et y (dont chacun prend les valeurs 0 ou 1), les valeurs de $x \wedge y$, $x \vee y$, $x \sim y$, $x \rightarrow y$ respectivement. Ces opérations s'appellent respectivement *multiplication logique*, *addition logique*, *équivalence et implication*. Elles sont complètement décrites par les tables 1.12 à 1.15 qu'on déduit de la table 1.11.

1.4.3. Propriétés fondamentales des opérations et des connecteurs. Deux propositions sont dites équivalentes si elles sont vraies ou fausses toutes les deux. L'équivalence de deux propositions est désignée par le signe d'égalité.

Convenons de coder les valeurs logiques **vrai** et **faux** respectivement par les nombres 1 et 0, et de désigner par une même lettre une proposition élémentaire et sa valeur de vérité. Une fois cette convention admise, on établit une correspondance biunivoque entre les équivalences du calcul des propositions et les égalités du calcul booléen qui permet de remplacer l'étude des propositions composées par celle des opérations logiques. Il est alors évident que la notation $A = B$ peut être lue de deux façons: a) les propositions A et B sont équivalentes; b) les nombres A et B sont égaux. Tout à fait de même, la notation $A = 1$ ($A = 0$) a deux interprétations: a) la proposition A est vraie (fausse); b) le nombre A vaut 1 (vaut 0).

Les propriétés de l'inversion (et de la négation) résultent des formules (1.5) et (1.6):

$$\bar{\bar{A}} = \overline{(\bar{A})} = A, \tag{1.7}$$

ce qui, en langage du calcul des propositions, veut dire: une double négation d'une proposition est équivalente à cette proposition.

Les propriétés de la multiplication logique (et de la conjonction) résultent immédiatement de la table 1.12:

$$\left.\begin{aligned} A \wedge B &= B \wedge A, \\ A \wedge 0 &= 0, \\ A \wedge 1 &= A, \\ A \wedge A &= A, \\ A \wedge \bar{A} &= 0, \\ A \wedge (B \wedge C) &= (A \wedge B) \wedge C. \end{aligned}\right\} \tag{1.8}$$

La première formule (1.8) signifie que la multiplication logique (et la conjonction) est commutative. La dernière formule (1.8) montre que la multiplication logique (et la conjonction) est associative, d'où l'autorisation d'omettre les parenthèses dans l'écriture des conjonctions à plusieurs éléments (par exemple, on peut écrire $A \wedge B \wedge C$).

Les propriétés de l'addition logique (et de la disjonction) se déduisent de la table 1.13:

$$\left.\begin{aligned} A \vee B &= B \vee A, \\ A \vee 0 &= A, \\ A \vee 1 &= 1, \\ A \vee A &= A, \\ A \vee \bar{A} &= 1, \\ A \vee (B \vee C) &= (A \vee B) \vee C. \end{aligned}\right\} \tag{1.9}$$

La première formule (1.9) traduit la loi commutative de l'addition logique (et de la disjonction), la dernière formule exprime l'associativité de cette opération ; on peut donc omettre les parenthèses dans les disjonctions à plusieurs éléments (par exemple, on peut écrire $A \vee B \vee C$).

Les formules

$$A \wedge B = \overline{\bar{A} \vee \bar{B}}, \tag{1.10}$$

$$A \vee B = \overline{\bar{A} \wedge \bar{B}} \tag{1.11}$$

sont d'une grande importance. Elles permettent de passer de la multiplication logique à l'addition logique et vice versa.

Enfin, signalons les formules

$$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C),$$

$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C).$$

La première formule signifie que la multiplication logique est distributive par rapport à l'addition logique (en algèbre élémentaire, son analogue est la relation $a(b + c) = ab + ac$). La seconde formule exprime la distributivité de l'addition logique par rapport à la multiplication logique (il n'y a pas d'analogue de cette formule en algèbre élémentaire, puisque l'égalité $a + bc = (a + b)(a + c)$ est fausse).

Pour l'implication et l'équivalence, on a les relations:

$$A \rightarrow B = \bar{A} \vee B,$$

$$A \sim B = (A \rightarrow B) \wedge (B \rightarrow A),$$

$$A \sim B = (\bar{A} \vee B) \wedge (A \vee B).$$

Signalons l'opération (et le connecteur) par laquelle on obtient $\overline{A \sim B}$ (la négation de l'équivalence) à partir de A et B. On désigne cette opération (et le connecteur) par le symbole spécial $\approx$ (se lit: non équivalent) ; on a par définition

$$A \approx B = \overline{A \sim B}. \tag{1.12}$$

L'opération de non-équivalence est complètement décrite par la table 1.16 qui se déduit aisément des tables 1.14 et 1.10.

Table 1.16

Non-équivalence

$0 \approx 0 = 0$
$0 \approx 1 = 1$
$1 \approx 0 = 1$
$1 \approx 1 = 0$

La fonction $x \approx y$ n'est rien d'autre que la fonction $F_7(x, y)$ de la table 1.11. Le symbole $\approx$ a le sens de « ou » exclusif, pour cette raison la proposition composée $A \approx B$ peut être lue : *ou bien A, ou bien B.*

Enfin, signalons la *fonction de Sheffer* ou la fonction incompatibilité logique qui représente la négation de la conjonction et qui se désigne par une barre inclinée « / » (on lit : *incompatible*) :

$$A/B = \overline{A \wedge B}. \tag{1.13}$$

La table de l'opération de Sheffer (table 1.17) se déduit facilement des tables 1.12 et 1.10.

Table 1.17

Opération de Sheffer

0/0 = 1
0/1 = 1
1/0 = 1
1/1 = 0

La fonction x/y n'est rien d'autre que la fonction $F_{15}(x, y)$ de la table 1.11.

En remplaçant dans (1.13) B par A on obtient après quelques simplifications :

$$\overline{A} = A/A, \tag{1.14}$$

et des formules (1.13) et (1.14) on tire

$$A \wedge B = (A/B)/(A/B). \tag{1.15}$$

Pour réduire le nombre de parenthèses dans des formules, on introduit un certain ordre de priorité d'opérations logiques et respectivement des connecteurs. On convient d'attribuer au signe $\vee$ un ordre de priorité plus élevé que celui des signes $\sim$, $\approx$, $\rightarrow$, /, la priorité du signe $\wedge$ est supérieure à celles des signes mentionnés (y compris $\vee$), le signe $\overline{}$ a la priorité la plus élevée.

D'après cette convention, on peut par exemple simplifier la notation

$$(A \vee (B \wedge C)) \rightarrow ((A \wedge B) \vee (C \wedge D))$$

comme suit

$$A \vee B \wedge C \rightarrow A \wedge B \vee C \wedge D.$$

1.4.4. Systèmes complets d'opérations et de connecteurs. Soit $\theta(x_1, x_2, \ldots, x_n)$ une fonction logique quelconque d'un nombre fini de variables binaires. Si θ n'est pas identiquement nulle, et si

$Q_1, Q_2, \ldots, Q_k$ sont tous les points de son domaine de définition où $\theta = 1$, alors on a la formule (v. [50], pp. 58 à 60):

$$\theta(x_1, x_2, \ldots, x_n) = \varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_k, \tag{1.16}$$

où

$$\varphi_j = y_{j,1} \wedge y_{j,2} \wedge \ldots \wedge y_{j,n} \qquad (j = 1, 2, \ldots, k),$$

avec

$$y_{j,i} = \begin{cases} x_i \text{ si } x_i = 1 \text{ au point } Q_j; \\ \bar{x}_i \text{ si } x_i = 0 \text{ au point } Q_j \end{cases}$$

pour $i = 1, 2, \ldots, n$.

De même, si la fonction θ n'est pas identiquement égale à l'unité et si $R_1, R_2, \ldots, R_l$ sont tous les points de son domaine de définition dans lesquels $\theta = 0$, alors on a la formule (v. [50], pp. 59 à 60):

$$\theta(x_1, x_2, \ldots, x_n) = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_l, \tag{1.17}$$

où

$$\psi_j = z_{j,1} \vee z_{j,2} \vee \ldots \vee z_{j,n} \qquad (j = 1, 2, \ldots, l),$$

avec

$$z_{j,i} = \begin{cases} x_i \text{ si } x_i = 0 \text{ au point } R_j; \\ \bar{x}_i \text{ si } x_i = 1 \text{ au point } R_j \end{cases}$$

pour $i = 1, 2, \ldots, n$.

Il résulte des formules (1.16) et (1.17) que toute fonction logique d'un nombre fini de variables binaires peut être exprimée par un nombre fini d'opérations $^-$, $\wedge$, $\vee$. C'est en ce sens que le système des opérations $^-$, $\wedge$, $\vee$ est dit *complet*.

La formule (1.11) exprime l'opération $\vee$ par les opérations $^-$ et $\wedge$, donc le système des opérations $^-$ et $\wedge$ est aussi complet. La formule (1.10) exprime l'opération $\wedge$ par $^-$ et $\vee$, donc le système des opérations $^-$ et $\vee$ est complet. Les formules (1.14) et (1.15) expriment les opérations $^-$ et $\wedge$ par seule l'opération de Sheffer, de sorte que cette opération constitue à elle seule un système complet.

Les formules (1.16) et (1.17) permettent d'exprimer toute fonction logique d'un nombre fini de variables binaires, donnée par une table, au moyen des opérations $^-$, $\wedge$ et $\vee$.

EXEMPLE 1.23. Soit $\theta(A, B, C)$ une fonction donnée par la table 1.18. Il faut l'exprimer par les opérations $^-$, $\wedge$, $\vee$.

Première méthode. Cherchons les points Q_i en lesquels $\theta(A, B, C) = 1$. On a $Q_1(0, 0, 1)$, $Q_2(1, 1, 0)$, $Q_3(1, 1, 1)$. Construisons φ_i $(i = 1, 2, 3)$:

$$\varphi_1 = \bar{A} \wedge \bar{B} \wedge C,$$
$$\varphi_2 = A \wedge B \wedge \bar{C},$$
$$\varphi_3 = A \wedge B \wedge C.$$

Table 1.18

Fonction θ (A, B, C)

A	B	C	θ
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Nous obtenons d'après la formule (1.16)

$$\theta = \bar{A} \wedge \bar{B} \wedge C \vee A \wedge B \wedge \bar{C} \vee A \wedge B \wedge C.$$

En simplifiant on a

$$\theta = \bar{A} \wedge \bar{B} \wedge C \vee A \wedge B.$$

Deuxième méthode. Nous trouvons les points R_1 (0, 0, 0); R_2 (0, 1, 0); R_3 (0, 1, 1); R_4 (1, 0, 0); R_5 (1, 0, 1) en lesquels θ (A, B, C) = 0. Construisons les fonctions ψ_i (i = 1, 2, 3, 4, 5):

$$\psi_1 = A \vee B \vee C, \quad \psi_3 = A \vee \bar{B} \vee \bar{C},$$
$$\psi_2 = A \vee \bar{B} \vee C, \quad \psi_4 = \bar{A} \vee B \vee C,$$
$$\psi_5 = \bar{A} \vee B \vee \bar{C}.$$

La formule (1.17) nous conduit à une autre expression pour θ (A, B, C):

$$\theta = (A \vee B \vee C) \wedge (A \vee \bar{B} \vee C) \wedge (A \vee \bar{B} \vee \bar{C}) \wedge$$
$$\wedge (\bar{A} \vee B \vee C) \wedge (\bar{A} \vee B \vee \bar{C}).$$

En simplifiant on obtient

$$\theta = (B \vee C) \wedge (A \vee \bar{B}) \wedge (\bar{A} \vee B \vee \bar{C}).$$

Les deux expressions de θ transforment l'une dans l'autre.

1.4.5. Notion de prédicat. Désignons par M l'ensemble de symboles deux à deux différents qui désignent les constructions particulières à partir desquelles on forme, moyennant des connecteurs, des propositions élémentaires. Formons des collections ordonnées de n symboles ($n \geqslant 1$) appartenant à l'ensemble M. On accepte des répétitions d'un même symbole dans une collection. Soit N un ensemble de collections mentionnées. Une collection arbitraire-

ment fixée de N sera notée $(a_1, a_2, \ldots, a_n)$, où a_i $(i = 1, 2, \ldots, n)$ désignent des symboles.

Soit $x_1, x_2, \ldots, x_n$ une suite de symboles deux à deux distincts qui n'appartiennent pas à M. Désignons-la par la lettre S et appelons *ensemble des arguments* (chaque symbole x_i sera donc appelé *argument*).

Considérons une configuration $\Phi(x_1, x_2, \ldots, x_n)$ de symboles appartenant à M ou à S qui satisfait aux conditions suivantes:

1) au moins un symbole de S figure dans $\Phi(x_1, x_2, \ldots, x_n)$;

2) si $\Phi(a_1, a_2, \ldots, a_n)$ est une collection quelconque de symboles qui fait partie de N, alors, en remplaçant les x_i par les symboles a_i partout dans $\Phi(x_1, x_2, \ldots, x_n)$, nous obtenons à partir de $\Phi(x_1, x_2, \ldots, x_n)$ une notation d'une proposition $\Phi(a_1, a_2, \ldots \ldots, a_n)$.

La configuration de symboles $\Phi(x_1, x_2, \ldots, x_n)$ s'appelle *relation*, et l'ensemble N, son *domaine de définition*. Si dans toutes les collections appartenant à N la i-ème place est occupée par les symboles qui sont des noms d'objets, alors l'argument x_i s'appelle *variable individuelle*. Dans ce qui suit on considère seules les relations dont tous les arguments sont des variables individuelles.

EXEMPLE 1.24. Soit M l'ensemble des mots français. Supposons qu'on en forme les collections suivantes contenant chacune un seul mot: « lion », « loup », « bœuf », « cerf », « renard », « écureuil ». L'ensemble N se compose des collections citées. La configuration de mots « le x est un herbivore » représente une relation. En remplaçant le symbole x par le mot « lion » on obtient une proposition fausse: « le lion est un herbivore »; en remplaçant le symbole « x » par le mot « bœuf » on obtient une proposition vraie: « le bœuf est un herbivore ».

EXEMPLE 1.25. Cette fois les éléments de l'ensemble M sont les entiers décimaux, les signes des opérations algébriques élémentaires et les signes $=, >, \geqslant, <, \leqslant$. Alors la configuration de symboles $x > 2y$ sera une relation dont le domaine de définition N est formé de tous les couples de nombres entiers. En choisissant le couple de nombres 2; 5 et en remplaçant x par 2 et y par 5, on arrive à une proposition fausse $2 > 2\cdot 5$ (se lit: deux est plus grand que deux fois cinq). En choisissant dans N le couple de symboles 5; 2 et en remplaçant x par 5 et y par 2, on obtient une proposition vraie $5 > 2\cdot 2$ (cinq est plus grand que deux fois deux).

La configuration

$$\frac{x+y}{1+x^2+y^2} > 3$$

est encore une relation.

Si une collection ordonnée $(a_1, a_2, \ldots, a_n)$ fait d'une relation $\Phi(x_1, x_2, \ldots, x_n)$ une proposition vraie, on dit que la relation $\Phi(x_1, x_2, \ldots, x_n)$ est *satisfaite* par $(a_1, a_2, \ldots, a_n)$. Et si la collection mentionnée transforme la relation en une proposition fausse, on dit que $(a_1, a_2, \ldots, a_n)$ *ne satisfait pas* à la relation.

On appelle *prédicat*

1) toute relation Φ;

2) chacune des notations $(\overline{\Phi})$, $(\Phi) \wedge (F)$, $(\Phi) \vee (F)$, $(\Phi) \rightarrow (F)$, $(\Phi) \sim (F)$, $(\Phi) \approx (F)$, où Φ et F sont des prédicats.

Si Φ et F ne contiennent pas de connecteurs logiques, on peut omettre les parenthèses dans les notations du point 2).

Deux prédicats sont dits *équivalents* s'ils ont le même domaine de définition et si toute collection ordonnée appartenant à ce domaine de définition vérifie simultanément ou non les deux prédicats.

EXEMPLE 1.26. Revenons aux ensembles M et N de l'exemple 1.25. La configuration

$$\overline{x > y \vee x = y}$$

est un prédicat. Pour les mêmes M et N,

$$x < y$$

est aussi un prédicat, les deux étant équivalents.

Faisons correspondre à chaque collection $(a_1, a_2, \ldots, a_n)$ de N le nombre 1, si elle vérifie une relation $\Phi(x_1, x_2, \ldots, x_n)$, et le nombre 0 dans le cas contraire. Nous définissons ainsi une fonction logique $F(x_1, x_2, \ldots, x_n)$ sur N. Une telle fonction logique correspond à un prédicat, de même qu'à une proposition correspond une valeur d'une certaine fonction logique de variables binaires.

§ 1.5. Eléments de la théorie des algorithmes

1.5.1. Notion d'algorithme. La notion d'algorithme a apparu en mathématiques en rapport avec les recherches des méthodes générales de résolution de problèmes d'un même type. On peut décrire cette notion de la manière suivante.

Un *algorithme* est une *prescription* (un ordre ou un système d'ordres) qui détermine l'enchaînement d'opérations élémentaires permettant d'obtenir, à partir des données initiales, le résultat cherché et qui possède les propriétés suivantes :

a) la netteté (i.e. une précision qui ne laisse aucune place à l'arbitraire et une clarté); grâce à cette propriété la réalisation d'un algorithme est un procédé mécanique;

b) l'efficacité, i.e. la propriété de conduire, dans les cas pour lesquels l'algorithme fut créé, au résultat cherché après un nombre fini d'opérations suffisamment simples;

c) de plus, la tradition veut qu'un bon algorithme soit universel, i.e. qu'il soit applicable à tout problème d'une certaine classe.

Il résulte du point b) que le processus de réalisation d'un algorithme doit être discret, se composer d'opérations élémentaires. L'exigence que ces opérations soient simples est liée au fait que, si on permet des opérations arbitrairement complexes, la notion d'algorithme ne sera plus déterminée du tout. Dans ce cas, chaque demande de résoudre un problème pourra être considérée comme un algorithme qui, de plus, détermine le processus réalisable en une seule opération, pourvu que la solution du problème existe. La propriété de conduire à la solution en un nombre fini de pas s'appelle *réalisabilité* (à condition, bien entendu, de limiter la complexité des opérations).

La formulation donnée plus haut de la notion d'algorithme est loin d'être rigoureuse et doit être considérée comme une simple description. En effet, on ne définit pas en mathématiques la notion de « prescription », de netteté, de clarté, de complexité limitée d'opérations. Néanmoins, on peut dégager certaines conclusions même d'une telle description. Le mot « prescription » veut dire proposition dans une certaine langue. Comme on a vu dans le § 1.2, les langues naturelles conviennent peu pour exprimer les prescriptions bien déterminées et précises. Par conséquent, un algorithme doit être une proposition dans un langage formel. Pour que cette proposition soit bien une prescription, un ordre, le langage formel doit posséder une sémantique (deuxième cas de définition d'une sémantique qu'on a décrit au p. 1.2.2). La netteté suppose également qu'on soit en mesure de juger de l'applicabilité d'un algorithme, étant donnée une information initiale (par exemple, s'il s'agit d'un algorithme d'addition des nombres entiers et d'un triangle comme donnée initiale, il est clair que l'algorithme ne convient pas). Le problème sera résolu, si à chaque algorithme correspond un langage formel de ses données initiales. Si nous exigeons que pour toute proposition du langage des données initiales l'algorithme possède la propriété de réalisabilité, le problème de construction d'un tel langage sera rendu extrêmement difficile. De plus, nombre de classes de problèmes comportent des problèmes n'ayant pas de solution. Pour cette raison, il est naturel d'accepter que pour certaines données initiales l'algorithme est réalisable, et pour certaines autres il ne l'est pas (exactement pour celles qui correspondent aux problèmes sans solution). La non-réalisabilité peut s'exprimer par ce que le processus algorithmique durera infiniment ou s'interrompra sans résultat.

Les conclusions que nous venons de faire s'appuient non seulement sur la description donnée plus haut de la notion d'algorithme, mais aussi sur quelques autres connaissances (notamment, sur les propriétés des langages naturels et formels), et, en fait, représentent déjà une précision de cette description. Pour que les algorithmes deviennent l'objet d'étude, il faut passer de la description en question à une définition rigoureuse. Comme une première étape dans la voie de précision on peut se borner à une description détaillée de certains cas spéciaux. Voyons, par exemple, l'algorithme d'Euclide bien connu.

EXEMPLE 1.27. L'algorithme d'Euclide sert à trouver le plus grand commun diviseur de deux nombres entiers positifs. En désignant par x et y deux nombres entiers positifs et par z le résultat cherché, et en supposant qu'on sait effectuer l'addition et la soustraction des entiers et l'opération de calcul du prédicat $P(a, b)$défini par la condition

$$P(a, b) = \begin{cases} 1 \text{ pour } a > b, \\ 0 \text{ pour } a \leqslant b, \end{cases}$$

on peut représenter l'algorithme d'Euclide de la manière suivante:

1°. Calculer la quantité

$$\omega_1 = P(x, y).$$

Aller au point numéro $2\omega_1 + 2$.

2°. Calculer la quantité

$$\omega_2 = P(y, x).$$

Aller au point numéro $2\omega_2 + 3$.

3°. Poser

$$z = x.$$

Saut au p. 6°.

4°. Calculer la différence

$$x - y$$

et considérer cette quantité comme x dans ce qui suit. Retour au p. 1°.

5°. Calculer la différence

$$y - x$$

et considérer cette quantité comme y dans ce qui suit. Retour au p. 1°.

6°. Arrêter les calculs.

L'universalité de l'algorithme d'Euclide consiste en ce qu'il est applicable à *n'importe quel* couple de nombres entiers positifs. Son efficacité consiste en ce qu'il détermine le procédé qui, pour tout couple d'entiers positifs, conduit à l'*obtention* de leur plus

grand commun diviseur. La netteté de l'algorithme d'Eclide est assurée par les faits que le réalisateur *sait* effectuer les opérations nommées dans l'algorithme, les *comprend* d'une manière univoque et, de plus, il sait par où commencer la réalisation de l'algorithme (par le p. 1°).

Sur l'exemple 1.27, nous voyons que les algorithmes du type d'algorithme d'Euclide se composent des prescriptions élémentaires dont chacune détermine deux processus: un processus principal qui est une transformation des opérandes, et un processus de passage à une autre prescription élémentaire. Dans certains cas particuliers les deux processus (ou l'un d'eux) s'avèrent nuls. Un processus principal nul représente une transformation identique, et un passage nul signifie la fin du processus. De plus, l'exemple 1.27 montre que pour réaliser un algorithme il faut connaître les règles de réalisation.

Conformément à ce qu'on a dit plus haut, nous choisissons pour première étape de précision de la notion d'algorithme la définition des algorithmes d'un type spécial que nous appelons algorithmes primaires.

1.5.2. Algorithmes primaires. Soient deux langages L_1 et L_2 dont on connaît que les constructions du langage L_2 sont des constructions bien déterminées d'une classe (A, B) et que le langage L_1 est engendré par un métalangage admettant les formules de Backus, ses phrases sont des mots dans un alphabet E.

Nous posons

$$E = E_1 \cup E_2 \cup E_3,$$

les alphabets E_1, E_2, E_3 étant deux à deux disjoints.

Supposons qu'à l'aide des formules de Backus soient définies les classes grammaticales suivantes: ⟨numéro⟩ qui est un mot dans E_1 et ⟨séparateur 1⟩, ⟨séparateur 2⟩, ⟨séparateur 3⟩, ⟨séparateur 4⟩, ⟨séparateur 5⟩, ⟨séparateur 6⟩, ⟨séparateur 7⟩ qui sont des mots dans E_2.

Pour fixer les idées, nous choisissons pour lettres de l'alphabet E_1 les chiffres arabes {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, mais prévenons d'avance que ce n'est pas obligatoire.

Supposons encore que, d'une manière ou d'une autre, on a défini les classes grammaticales ⟨signe d'action⟩ et ⟨signe de condition⟩ dont les éléments des mots dans E_3.

Posons:

⟨étiquette⟩ : : = ⟨numéro⟩

⟨renvoi⟩ : : = ⟨numéro⟩

⟨renvoi inconditionnel⟩ : : = ⟨renvoi⟩

⟨renvoi conditionnel⟩ : : = ⟨renvoi⟩ ⟨séparateur 7⟩ ⟨renvoi⟩

⟨signe d'opération⟩: : = ⟨signe d'action⟩ | ⟨signe de condition⟩
⟨ordre⟩: : = ⟨séparateur 1⟩ ⟨étiquette⟩ ⟨séparateur 2⟩ ⟨signe d'action⟩ ⟨séparateur 3⟩ ⟨renvoi inconditionnel⟩ ⟨séparateur 4⟩ | ⟨séparateur 1⟩ ⟨étiquette⟩ ⟨séparateur 2⟩ ⟨séparateur 3⟩ ⟨séparateur 4⟩ | ⟨séparateur 1) ⟨étiquette⟩ ⟨séparateur 5⟩ ⟨signe de condition⟩ ⟨séparateur 6⟩ ⟨renvoi conditionnel⟩ ⟨séparateur 4⟩ ⟨algorithme primaire⟩: : = ⟨ordre⟩ | ⟨algorithme primaire⟩ ⟨ordre⟩.

Un ordre ayant le renvoi et le signe d'action vides sera utilisé, on le verra plus tard, comme ordre d'arrêter le processus.

Supposons à présent qu'il y ait un ensemble H d'opérations *) sur les constructions de la classe (A, B); à chaque opération est mis en correspondance un signe d'opération et elle représente soit une action (alors il lui correspond un signe d'action), soit une condition (alors il lui correspond un signe de condition). On suppose que les actions transforment les constructions de la classe (A, B) en des constructions de la même ou d'une autre classe, tandis que les conditions, sans transformer les constructions, leur font correspondre des mots dans un alphabet Λ disjoint de l'alphabet $A \cup B$. Dans un cas spécial **) (auquel nous nous bornons dans ce qui suit) il n'existe que deux mots dans Λ. La vérification d'une condition donne l'un de ces mots, on peut les interpréter l'un comme « oui », l'autre comme « non ».

Désignons les signes concrets d'opérations (d'actions et de conditions) respectivement par les métasymboles

$$z_1, z_2, \dots, z_n, p_1, p_2, \dots, p_m,$$

et les opérations qui leur correspondent par les métasymboles

$$\tilde{z}_1, \tilde{z}_2, \dots, \tilde{z}_n, \dots, \tilde{p}_1, \tilde{p}_2, \dots, \tilde{p}_m$$

(ici z est employé pour désigner une action et p pour désigner une condition).

A présent, définissons la règle d'exécution d'un algorithme primaire, étant donné son opérande (qu'on appelle donné initiale) représentant une construction du langage L_2. Remarquons qu'une langue courante est mal adaptée à la formulation de telles règles, pour cette raison il nous faudra introduire quelques nouveaux termes. La règle contiendra des indications concernant la réalisation des actions sur les éléments de la règle elle-même, sur les éléments de l'algorithme primaire et sur ceux de la construction transformée. Les actions sur les éléments de la règle se réduiront à des passages de l'un de ses points à un autre ou à l'arrêt du processus. Convenons

*) Les notions d'opération, d'action et de condition seront précisées plus bas (v. p. 1.5.3).

**) En général, on peut envisager des cas où le nombre de mots dans Λ est supérieur à deux et même infini.

de formuler l'ordre de passage à un point de numéro j en termes suivants: « aller en j »; l'ordre de cesser le processus désignons par le mot « fin ».

Les actions sur les éléments de l'algorithme seront classées en deux groupes, et on évitera de confondre des actions d'un même type appartenant aux groupes différents. Pour cette raison, la recherche du début d'un algorithme sera désignée par les mots « trouver le début de l'algorithme » si elle fait partie du premier groupe d'actions, et par les mots « fixer le début de l'algorithme » si elle appartient au deuxième groupe.

Une progression vers la droite le long d'une chaîne de notations d'un élément trouvé à un élément à trouver sera désignée par les mots « passer à ... ». On considérera le nouveau élément comme trouvé, et le précédent ne sera plus considéré. Le déplacement de ce qu'on a déjà fixé vers un élément disposé plus à droite sera désigné par les mots « se déplacer vers... ». Dans ce cas, le nouveau élément sera considéré comme fixé (et celui qu'on a fixé avant ne sera plus considéré). Le déplacement d'un élément obtenu par une action du premier groupe à un élément obtenu par une action du deuxième groupe sera désigné par les mots « passer à ... fixé ». Nous convenons qu'après l'exécution de cet ordre, il y aura l'élément trouvé et il n'y aura aucun élément fixé.

Règle W d'exécution d'un algorithme primaire

1°. Trouver la première étiquette (à compter du début); aller en 2°.

2°. Vérifier la condition: « le signe d'opération dans l'ordre dont la marque est trouvée n'est pas vide ». Si oui, aller en 9°, si non, aller en 10°.

3°. Passer au signe de renvoi; aller en 4°.

4°. Passer au signe de renvoi; aller en 5°.

5°. Fixer la première étiquette; aller en 6°.

6°. Vérifier si le renvoi trouvé et l'étiquette prise sont identiques; si oui, aller en 7°, si non, aller en 8°.

7°. Passer à l'étiquette fixée; aller en 3°.

8. Se déplacer vers l'étiquette suivante; aller en 6°.

9. Passer au signe d'opération; aller en 11°.

10. Fin.

11. Vérifier la condition: « le signe d'opération trouvé est identique à z_1 »; si oui, aller en 12°, si non, aller en 13°.

12 . Effectuer l'opération $\tilde{z}_1$ sur la construction à traiter; aller en 4°.

$[11 + 2(n - 1)]^{o}$. Vérifier la condition: « le signe d'opération trouvé est identique à z_n »; si oui, aller en $[11 + 2(n - 1) + 1]^{o}$, si non, aller en $[11 + 2n]^{o}$.

$[11 + 2(n - 1) + 1]^{o}$. Effectuer l'opération $\tilde{z}_n$ sur la construction à traiter; aller en 4°.

$[11 + 2n]^{o}$. Vérifier la condition: « le signe d'opération trouvé est identique à p_1 »; si oui, aller en $[11 + 2n + 1]^{o}$, si non, aller en $[11 + 2n + 2]^{o}$.

$[11 + 2n + 1]^{o}$. Vérifier si la condition $\tilde{p}_1$ est satisfaite par la construction transformée; si oui, aller en 4°, si non, aller en 3°.

. . . .

$[11 + 2n + 2(m - 1) - 2]^{o}$. Vérifier la condition: « le signe d'opération trouvé est identique à p_{m-1} »; si oui, aller en $[11 + 2n + 2(m - 1) - 1]^{o}$, si non, aller en $[11 + 2n + 2(m - 1)]^{o}$.

$[11 + 2n + 2(m - 1) - 1]^{o}$. Vérifier si la condition $\tilde{p}_{m-1}$ est satisfaite par la construction transformée; si oui, aller en 4°, si non, aller en 3°.

$[11 + 2n + 2(m - 1)]^{o}$. Vérifier si la condition $\tilde{p}_m$ est satisfaite par la construction transformée; si oui, aller en 4°, si non, aller en 3°.

Soulignons que nous avons formulé la règle W dans une langue naturelle formalisée.

Chaque mot ⟨algorithme primaire⟩ du langage L_1 considéré avec la règle W sera appelé *algorithme primaire* ou *algorithme primaire dans le langage* L_1 *sur le langage* L_2.

On voit de ce qui précède qu'un algorithme primaire peut être appliqué, à l'aide de la règle W, à une construction du langage L_2. Le processus de réalisation de l'algorithme peut soit conduire à un résultat déterminé (lorsqu'il s'achève par l'exécution du p. 10° de la règle W), soit durer indéfiniment, soit enfin s'interrompre sans résultat (s'il est impossible d'appliquer à un résultat intermédiaire l'opération imposée par l'algorithme ou si en exécutant le p. 8°) on ne trouve pas de marque cherchée.

Dans le premier cas on dit que l'algorithme primaire est *applicable* à l'opérande qui est la donnée initiale et dans les deux autres, *non applicable*.

1.5.3. Notion d'opération. Dans la notion d'algorithme primaire nous avons utilisé celle d'opération. Nous nous sommes servi de cette notion au § 1.2 en parlant des langages formels. Il est évident que, pour préciser les notions de langage et d'algorithme primaire, il faut attribuer à celle d'opération un sens rigoureux.

Nous partons du fait qu'il existe des opérations qu'on peut qualifier comme opérations de base (non définissables). A partir de

celles-ci, on construit des entités qu'on *déclare opérations*. C'est de cette manière qu'on obtient toutes les opérations.

Ceci dit, introduisons la notion d'opération de la façon suivante. Nous distinguons deux types d'*opérations*: les *actions* et les *conditions*.

On appelle actions

1a) les actions naturelles (v. plus bas);

2a) la linéarisation et la délinéarisation;

3a) toute application, déclarée action, réalisée par un algorithme qui opère sur les constructions du langage L_2 (en particulier, des collections ordonnées de r constructions ($r \geqslant 0$) sont des constructions);

4a) il n'existe pas d'autres actions.

Notons qu'une action suppose le remplacement d'une construction initiale par le résultat d'action (traitement de la construction initiale). Lorsque les données initiales d'une action sont des collections de r constructions ($r \geqslant 0$), et que $r =$ const, alors r s'appelle *rang de l'action*.

On appelle conditions

1b) les conditions naturelles (v. plus bas);

2b) toute application, déclarée condition, réalisée par un algorithme qui transforme les constructions du langage L_2 (en particulier, des collections ordonnées de r constructions) en des mots dans l'alphabet Λ qui n'a pas de partie commune avec l'alphabet du langage L_2;

3b) il n'existe pas d'autres conditions.

Lorsque les données initiales sont des collections de r constructions ($r \geqslant 0$), et que $r =$ const, alors r s'appelle *rang d'une condition*.

La vérification d'une condition ne suppose pas le remplacement de la construction initiale par le résultat de la vérification.

Les actions naturelles et les conditions naturelles sont appelées opérations naturelles.

Les données initiales des opérations naturelles sont ou bien des mots, ou bien des constructions spéciales dont chacune s'obtient à partir d'un mot en liant l'une de ses lettres par un connecteur spécial de rang un, qui n'est ni connecteur de début ni de fin. On peut appeler de telles constructions *mots à lettre distinguée*. Les résultats d'actions naturelles sont des mots ou des mots à lettre distinguée. Les résultats de vérification des conditions naturelles sont des valeurs de vérité (des mots dans l'alphabet Λ qui n'appartiennent pas à L_2, comme on l'a déjà dit dans la définition des conditions).

Les actions naturelles sont de quatre groupes. Leur sens est décrit par des propositions en mode impératif correspondantes (v. la table 1.19).

Le groupe I comprend les actions naturelles qui transforment un mot en un mot, et l'opération de génération de mots qui transforme un ensemble de mots vide en un mot vide.

Le groupe II comprend une action naturelle qui transforme un mot en un mot à lettre distinguée.

Table 1.19

Liste des actions naturelles

Numéro du groupe	Nom de l'action	Acte à réaliser
I	Génération d'un mot	Commencer à considérer un mot vide
	α-génération	A la place d'un mot donné vide, considérer un mot à une lettre, cette lettre étant α
	Annihilation	A la place d'un mot donné à une lettre considérer un mot vide
II	Recherche du début d'un mot	Trouver (et marquer par le connecteur spécial) le début du mot considéré
III	Progression	Passer de la lettre considérée d'un mot à la lettre immédiatement suivante (si la lettre considérée est une lettre finale, l'exécution de l'action est impossible)
	Régression	Passer de la lettre considérée d'un mot à la lettre immédiatement précédente (si l'on considère le début du mot, l'exécution de l'action est impossible)
	Rallongement (sans déplacement)	A la suite de la lettre considérée d'un mot écrire une lettre identique (l'exécution de l'action n'est possible que lorsqu'on considère la fin du mot)
	Rejet de la fin	Considérer la lettre traitée d'un mot comme fin du mot
	Remplacement d'une lettre par α	Remplacer la lettre considérée d'un mot par la lettre α
IV	Abandon d'une lettre	Cesser la considération d'une lettre isolée du mot donné

Le groupe III réunit les actions naturelles qui transforment un mot à lettre distinguée en un mot à lettre distinguée.

Enfin, le groupe IV contient une action naturelle qui transforme un mot à lettre distinguée en un mot ordinaire.

Les conditions naturelles sont au nombre de quatre *). La première concerne les mots ordinaires, les trois autres les mots à lettre distinguée. On peut formuler les conditions naturelles de la manière suivante :

a) « le mot considéré n'est pas vide »;

b) « la lettre considérée est le début d'un mot donné »;

c) « la lettre considérée est la fin d'un mot donné »;

d) « la lettre considérée est identique à la lettre donnée α ».

Appelons ces prédicats respectivement *condition « non vide »*, *condition de début*, *condition de fin* et *condition d'identité de lettres.*

Au § 1.2 nous avons considéré quelques opérations (de liaison de mots) que l'on peut maintenant définir, de façon formelle, à partir du p. 3a) de la définition d'une action. Tout ce qu'on a dit là peut être répété pour ' · substitutions markoviennes (voir § 5.5).

1.5.4. Algorithmes naturels. Supposons que L_2 est un langage de chaînes avec une grammaire inductive (et avec l'alphabet A) et qu'en donnant son alphabet on définit la classe grammaticale

⟨lettre dans A⟩.

Supposons qu'en plus de ce qu'on a dit, pour donner L_2 une seule formule de Backus suffit :

⟨mot dans A⟩ : : = ⟨lettre dans A⟩ | ⟨mot dans A⟩⟨lettre dans A⟩.

Le langage obtenu L_2 sera appelé langage des opérandes naturels.

Concrétisons le langage L_1 de la manière suivante. Posons que l'alphabet E_2 soit de la forme

$$E_2 = \{:,\ ;,\ /\}$$

et posons

⟨séparateur 1⟩ : : = ⟨vide⟩;
⟨séparateur 2⟩ : : = :
⟨séparateur 3⟩ : : = :
⟨séparateur 4⟩ : : = :
⟨séparateur 5⟩ : : = :
⟨séparateur 6⟩ : : = :
⟨séparateur 7⟩ : : = /

*) En fait, elles sont $3 + n$, où n est le nombre des lettres différentes α.

Ainsi, les ordres seront de la forme

⟨étiquette⟩: ⟨signe d'action⟩: ⟨renvoi⟩;

ou

⟨étiquette⟩::;

ou encore

⟨étiquette⟩: ⟨signe de condition⟩: ⟨renvoi⟩/⟨renvoi⟩;

En ce qui concerne l'alphabet E_3, nous posons

$$E_3 = E'_3 \cup A$$

(rappelons que A l'alphabet du langage L_2). Convenons de construire les signes d'opération sous la forme de mots dans E_3, en posant que chaque ⟨signe d'opération⟩ contient au moins une lettre dans E'_3. Pour E'_3 choisissons

$$E'_3 = \{s, h, \Delta, *, \rightarrow, \leftarrow, \nearrow, \searrow, !, \nabla, d, f, \equiv\}.$$

Prenons pour signes d'opération dont il s'agissait au p. 1.5.3 les mots dans l'alphabet E_3 indiqués dans la table 1.20.

Dans cette table α est un métasymbole et désigne une lettre arbitraire dans A.

Table 1.20

Signes des opérations naturelles

Nom de l'opération naturelle	Signe de l'opération
Génération d'un mot	s
α-génération	$h\alpha$
Annihilation	Δ
Recherche du début d'un mot	$*$
Progression	$\rightarrow$
Régression	$\leftarrow$
Rallongement	$\nearrow$
Rejet de la fin	$\searrow$
Remplacement d'une lettre par α	$!\alpha$
Abandon d'une lettre	$\times$
Condition « non vide »	∇
Condition de début	d
Condition de fin	f
Condition d'identité	$\equiv$
Action vide	⟨vide⟩

Une famille d'algorithmes primaires dont les opérations sont celles de la liste des opérations naturelles indiquées sera appelée famille d'*algorithmes naturels.*

Remarquons en anticipant que les algorithmes primaires, et donc les algorithmes naturels, sont un cas spécial de la notion d'algorithme. Aussi les algorithmes naturels peuvent servir de moyen à la construction de nouvelles opérations (autres que linéarisation, délinéarisation, opérations naturelles), ces dernières pouvant à leur tour servir de base à la construction de nouvelles classes d'algorithmes primaires et d'opérations.

Exemple 1.28. Un cas intéressant est fourni par les algorithmes naturels à l'aide desquels on peut définir des opérations de rang nul.

Pour l'exécution de tels algorithmes naturels, on n'a pas besoin de données initiales. Leur premier ordre est un ordre qui contient une action naturelle *s*.

Soient α_1 et α_2 des lettres de l'alphabet du langage des opérandes naturels. Considérons l'algorithme naturel :

$1 : s : 2;$	$6 : !\alpha_2 : 7;$
$2 : h\alpha : 3;$	$7 : \nearrow : 8;$
$3 : * : 4;$	$8 : \rightarrow : 9;$
$4 : \nearrow : 5;$	$9 : !\alpha_1 : 10;$
$5 : \rightarrow : 6;$	$10 : \times : 11;$
	$11 : \quad : ;$

Le résultat d'exécution de cet algorithme naturel est un mot $\alpha_1\alpha_2\alpha_1$. Ayant cet algorithme naturel (qui est un cas particulier de l'algorithme), nous sommes en droit de *déclarer* l'opération de rang nul qui a pour résultat le mot $\alpha_1\alpha_2\alpha_1$.

Exemple 1.29. En déclarant de nouvelles conditions (p. 1.5.3) on considère le langage d'opérandes élargi qui, en plus des constructions de la classe (A, B), contient un certain nombre de mots dans Λ. Avec un tel langage d'opérandes élargi, tout algorithme primaire qui transforme les constructions du langage L_2 en des mots dans Λ, peut être utilisé pour obtenir une nouvelle condition.

Par exemple, l'algorithme naturel

$1: \nabla \quad 2/5;$	$4: !\lambda_1 :6;$
$2: * \quad :3;$	$5: h\lambda_2 :6;$
$3: f \quad :4/8;$	$6: \times :7;$
$8: \searrow \quad :9;$	$7: \quad : ;$
	$9: !\lambda_2 :7;$

peut servir à la déclaration de la condition dont le sens est « le mot considéré à une seule lettre ». Pour désigner (nommer) cette condition, on peut choisir n'importe quel mot dans l'alphabet E_3, pourvu qu'il contienne des lettres dans E'_3.

En s'appuyant sur la définition des algorithmes primaires donnée au p. 1.5.2 et sur la définition de l'opération du p. 1.5.3, nous pouvons construire un grand nombre de classes d'algorithmes primaires.

1.5.5. Définition complète de l'algorithme. Pour formuler une définition complète de l'algorithme, nous commençons par sa définition récursive qui se base sur la notion d'algorithme primaire. C'est en réunissant toutes les définitions intimement liées l'une à l'autre (celles de l'algorithme, du langage, de l'opération) que nous arrivons enfin à une définition de l'algorithme.

Formulons donc la définition récursive de l'algorithme. Elle se compose de deux points.

Soient deux langages L_1 et L_2. Soit L_3 un langage dont les constructions sont les couples (t, s), où t est une construction du langage L_1 et s une construction du langage L_2. Alors

1. Tout algorithme primaire dans le langage L_1 sur le langage L_2 s'appelle *algorithme* dans le langage L_1 sur le langage L_2.

2. S'il y a un algorithme W (dans un langage quelconque) sur le langage L_3, alors toute construction t considérée avec W s'appelle *algorithme* dans le langage L_1 sur le langage L_2. La construction t est appelée *notation de l'algorithme.*

Ceci posé, toute construction s qui forme avec t une donnée initiale t, s pour W est appelée *donnée initiale* par rapport à l'algorithme noté t, et la construction r que l'on obtient en appliquant W à t, s s'appelle *résultat* (*cherché*) ; W s'appelle *algorithme d'exécution.*

Dans ce qui suit, si cela ne conduit pas à une confusion, nous dirons tout court « algorithme t » et « W-algorithme » au lieu d'« algorithme noté t » et « algorithme d'exécution W ».

Pour un algorithme donné dans un langage L_1 sur un langage L_2, nous appellerons L_1 *langage algorithmique* et L_2 *langage des données initiales.*

Dans un cas particulier, les résultats de travail d'un algorithme peuvent appartenir au langage L_2. Alors L_2 est appelé *langage des opérandes.*

La question se pose : comment est l'ensemble des résultats d'un algorithme sur le langage L_2 ? Pour un algorithme naturel, il est évident que les résultats forment un langage. Pour un algorithme primaire arbitraire, ce n'est plus évident.

Il s'avère tout de même que, si t est un algorithme sur le langage L_2, alors tous ses résultats appartiennent à un langage. Même plus,

il existe un langage contenant les resultats de l'algorithme t et ne contenant aucune autre construction.

1.5.6. Calculateur comme réalisation physique de l'algorithme d'exécution des programmes. La définition complète de l'algorithme permet d'élucider certaines notions de la théorie mathématique des machines commandées par programmes et de la programmation. Nous nous arrêtons pour le moment sur l'interprétation de la notion de calculateur, les autres problèmes seront traités dans les chapitres suivants.

Un calculateur réel peut être considéré comme une réalisation physique d'un calculateur abstrait. Le fonctionnement du calculateur est alors représenté comme une réalisation physique du processus d'exécution d'un algorithme W dont les données initiales sont des couples de mots t, s, où les mots t sont les programmes et les mots s, les données initiales. Une telle conception du calculateur permet de lui associer comme image mathématique un algorithme.

L'ensemble de tous les programmes possibles pour un calculateur donné représente son *langage d'algorithmes* et l'ensemble de toutes les données initiales et de tous les résultats possibles, le *langage des opérandes de la machine*. Donnons maintenant une interprétation mathématique de la mémoire d'un calculateur .

Un calculateur moderne possède un certain nombre d'organes d'entrée. Chacun d'eux permet d'introduire une suite linéaire de symboles, finie bien qu'arbitrairement longue (si on ne tient pas compte des dépenses du temps). Pour fixer les idées, nous nous bornerons au cas où l'information est stockée sur un support perforé. Nous avons donc, à l'entrée, un ensemble de mots $P_1, P_2, \ldots, P_{n_1}$ inscrits, par exemple sur le ruban perforé, n_1 étant le nombre d'organes d'entrée.

Il y a plusieurs types de mémoires : une mémoire principale à fonctionnement rapide et un nombre de mémoires auxiliaires (bandes, tambours, disques magnétiques) ; l'information est représentée sous forme de suite linéaires de symboles. Désignons ces suites (mots) par $Q_0, Q_1, \ldots, Q_{n_2}$, où Q_0 est le mot enregistré dans la mémoire rapide. Durant le fonctionnement du calculateur, le mot Q_0 conserve sa longueur égale au nombre d'éléments de mémoire rapide réservés à un mot. Les longueurs des autres Q_i sont bornées.

Le calculateur moderne possède divers types d'organes de sortie. L'information est délivrée sous forme de mots $R_1, R_2, \ldots, R_{n_3}$, où n_3 est le nombre d'organes de sortie. Les mots R_i, tout comme P_i, sont de longueur finie, mais arbitrairement grande.

Les mots mentionnés forment une construction au sens du p. 1.1.1 (voir fig. 1.3), qui est un modèle mathématique des mémoires du calculateur. Cette construction n'est pas une suite des mots en question. Tous les mots y sont directement liés au mot Q_0, les types de

liaisons dépendant des possibilités d'échange d'information entre les organes en question et, respectivement, entre les mots (voir fig. 1.3).

Au cours du fonctionnement du calculateur, on peut transférer l'information d'un mot P_i dans le mot Q_0 en « coupant » une partie

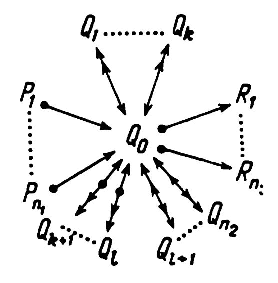


Fig. 1.3. Une construction correspondant à l'ensemble des mémoires d'un calculateur.

$P_l, \ldots, P_{nl}$ – données initiales;
$Q_l, \ldots, Q_k$ – information sur bandes magnétiques;
$Q_{k+1}, \ldots, Q_l$ – information sur disques magnétiques;
$Q_{l+1}, \ldots, Q_{n_2}$ – information sur tambours magnétiques;
$R_1, \ldots, R_{n_3}$ – résultats

initiale du mot P_i et en la substituant dans Q_0 à une partie de celui-ci de même longueur. Alors le mot P_i « se raccourcit ». La sortie de l'information du calculateur se fait en complétant des mots R_i par des morceaux du mot Q_0 sans effacer ces morceaux dans Q_0. Le transfert de l'information de Q_0 à P_i et de R_i à Q_0 est impossible.

Le transfert de l'information de Q_0 dans Q_i ($i > 0$) peut s'effectuer soit en substituant des morceaux du mot Q_0 (sans les effacer dans le mot Q_0), soit en complétant les mots Q_i par de tels morceaux, sans toutefois dépasser la longueur limite propre à chaque Q_i particulier.

A part les transferts d'information mentionnés, le traitement de la construction en question consiste en la transformation du mot Q_0, sans toucher à sa longueur. Les opérations effectuées sur Q_0 sont dans ce cas *localement données*. Il s'agit notamment d'*opérations élémentaires de la machine* qui s'effectuent sur certaines parties du mot Q_0 et dont le résultat est substitué à une partie de Q_0 ayant la même longueur que ce résultat.

Le modèle décrit du calculateur peut servir en cas de fonctionnement simultané de plusieurs organes d'entrée-sortie et d'un seul organe de calcul.

CHAPITRE 2

LANGAGE DES SCHÈMAS LOGIQUES ET LANGAGES DE LA MACHINE

§ 2.1. Langage des schémas logiques (YALS) *)

Le langage algorithmique des schémas logiques (YALS) a été créé pour la description de divers processus discrets. Son but était de faciliter la description de processus discrets et de rendre plus aisées les transformations équivalentes des algorithmes ainsi que de satisfaire à certains besoins de la programmation. Parmi tous les langages algorithmiques, le YALS est peut-être le plus proche du langage des descriptions mathématiques.

Le développement du YALS commence en 1953, lorsque A. Liapounov introduit les notions d'opérateurs du schéma de calcul et du schéma de programme [60]. En 1955, Yu. Yanov [94], en étudiant les schémas des algorithmes, formalise la description des opérateurs logiques dans le YALS. En 1959, N. Krinitski [35] propose une formalisation générale des opérateurs du langage des schémas logiques et définit une sémantique de ce langage (comme algorithme d'exécution d'un schéma logique). Le YALS trouve sa forme définitive dans l'ouvrage [43] de A. Liapounov et dans les ouvrages [27, 29, 32].

2.1.1. Alphabet du YALS. Dans la description de certains langages algorithmiques il est plus commode de se servir du terme « symbole de base » au lieu du terme « lettre » utilisé dans la théorie des algorithmes, en réservant le mot « lettre » à la désignation des lettres d'alphabets naturels. Pour ne pas confondre les métasymboles et les symboles du YALS, et pour séparer des textes écrits en YALS, nous utiliserons le signe || qui n'est pas un symbole du YALS.

Les symboles de base du YALS sont:

— les chiffres décimaux || 0|| 1|| 2|| 3|| 4|| 5|| 6|| 7|| 8|| 9 ||;

— les lettres latines minuscules italiques || a|| b|| c|| . . .|| y|| z||;

— les signes d'opérations et de fonctions employés en mathématiques (par exemple, || $+$|| $-$|| $:$|| sin || lg|| $\sqrt{}$ ||, etc.);

— les connecteurs logiques **) || $\wedge$|| $\vee$|| $\neg$|| $\supset$|| $\sim$||;

— les signes de relations || $>$|| $\geqslant$|| $<$|| $\leqslant$|| || $=$|| $\neq$|| ;

— les parenthèses || (||)||, les crochets || [||]||, les accolades

---

*) En russe: YAsyk Loguitcheskikh Skhem.

**) La barre habituelle exprimant la négation est remplacée par le signe ☐ écrit sur la ligne.

|| {|| }||, les flèches || ↓ || ↑||, de même que la virgule || ,|| et le point-virgule || ;||;

— les lettres latines majuscules italiques || *D*|| *F*|| *P*|| *Q*|| *S*||*V*|| et || *I*|| *E*|| *T*||;

— les demi-crochets: ouvrant supérieur || ⌜ ||, ouvrant inférieur || ⌞ || et fermant || ⌟ || (seulement inférieur), de même que le signe d'affectation || : = ||.

Les collections des symboles de base de tous les groupes sauf le dernier peuvent être complétées au besoin. L'ensemble des symboles de base signalés représente l'alphabet du langage YALS.

2.1.2. Structures primaires du langage des schémas logiques. Les morphèmes dans le YALS sont:

— les nombres naturels et les nombres;

— les cellules et les cellules dépendant de paramètres;

— les fonctions et les fonctions dépendant de paramètres;

— les signes de passage et les signes de passage dépendant de paramètres;

— les signes d'opérateurs et les signes d'opérateurs dépendant de paramètres, de même que les signes d'opérateurs généralisés et les signes d'opérateurs généralisés dépendant de paramètres.

Nombres naturels et nombres. On appelle nombre naturel toute suite finie de chiffres décimaux. Cette notation a le sens usuel. On appelle nombre soit un nombre naturel, soit une suite finie de chiffres précédée du signe + ou du signe — et séparée par une virgule en deux parties. On a à gauche la partie entière et à droite, la partie fractionnaire du nombre.

De la sorte, les morphèmes du YALS qu'on appelle nombres naturels et nombres ne sont rien d'autre que les notations décimales usuelles des nombres naturels et des nombres rationnels (positifs et négatifs).

Cellules. Classes de cellules utilisées dans le YALS. Si ξ est une lettre latine minuscule quelconque, sauf p et f, et $\nu_1, \nu_2, \ldots, \nu_k$ sont des nombres naturels, alors les notations de la forme

$$\xi \text{ et } \xi_{\nu_1, \nu_2, \nu_3, \ldots, \nu_k}$$

sont appelées *cellules.* Les indices sont figurés soit en caractères plus petits que la lettre principale de la notation, soit entre les flèches ↓ et ↑. Dans le dernier cas, on écrit

$$\xi \downarrow \nu_1, \nu_2, \ldots, \nu_k \uparrow .$$

L'ensemble de toutes les cellules utilisées pour chaque application particulière du YALS s'appelle *mémoire.*

Il ne faut pas comprendre les termes « cellule » et « mémoire » dans leur sens usité. Une cellule est une notation utilisée pour désigner une grandeur. Ce n'est que dans des cas particuliers que le

sens des notions de cellule et de mémoire coïncide avec leur sens ordinaire adopté dans la théorie des calculateurs et de la programmation (on préfère de dire « case » en parlant d'une cellule de mémoire de calculateur).

Les cellules utilisées en YALS sont les *objets*, les *paramètres* et les *cellules logiques*. La mémoire est respectivement constituée par la *mémoire des objets*, la *mémoire des paramètres* et la *mémoire logique*. Dans ce qui suit, nous convenons que la mémoire logique se compose d'une seule cellule logique. Pour désigner un objet on utilisera n'importe quelle minuscule sauf i, j, k, l, m, n, o, et sauf f et p ; pour noter un paramètre on choisira l'une des lettres i, j, k, l, m, n. La cellule logique sera toujours notée comme o (sans indices).

Par définition, on associe à chaque cellule un ensemble d'éléments appelés ses états possibles. Une cellule est caractérisée à tout moment par son état.

Les états des cellules paramètres sont des nombres naturels. Pour que notre terminologie ne diffère pas trop de la terminologie mathématique usitée, nous dirons « valeur d'un paramètre » au lieu d'« état d'un paramètre ».

La cellule logique peut avoir pour état les valeurs logiques **faux** et **vrai**. La valeur logique **faux** est désignée par le symbole $\|0\|$, et la valeur logique **vrai** est désignée par $\|1\|$. Les états des cellules objets dans le langage des schémas logiques sont formalisés. Pour toute application du YALS, on doit préciser ce que sont les états des objets. Lorsqu'on applique le YALS à la description d'une procédure de résolution d'un problème mathématique (problème de calcul), les objets seront des variables (si l'on ne fait pas de distinction entre un nom et ce qui est nommé), et leurs états seront des nombres (valeurs de ces variables). Si l'on applique le YALS à la description d'un processus physique discret, les objets pourront être les désignations de grandeurs physiques, leurs états seront alors les noms ou les descriptions des états de ces grandeurs physiques.

Remarquons qu'une cellule dont l'état est une constante peut être désignée par la notation de cet état permanent. On fera ainsi chaque fois quand on ne risque pas de confusions. Lorsque les états de certaines cellules sont des cellules (leurs noms), la simplification signalée n'est pas admissible. Au contraire, on l'utilise souvent pour les paramètres et pour les objets qui sont des êtres mathématiques. Par exemple, lorsqu'un paramètre i prend l'unique valeur 2, la cellule $\|i\|$ peut être désignée par le symbole $\|2\|$. Si au cours de la résolution d'un problème l'état d'une cellule $\|x\|$ est le nombre 3,217, alors la cellule $\|x\|$ peut être notée comme $\|3{,}217\|$. De telles notations sont commodes pour désigner dans le YALS les fonctions. Soulignons que l'utilisation de telles désignations pour les objets n'est possible qu'après une formalisation complète du YALS dépendant du domaine d'application de ce langage.

Opérations et fonctions. Classes des fonctions employées dans le YALS. Soit θ l'un quelconque des symboles

$$f, p, f_{v_1,v_2,\ldots,v_k}, p_{v_1,v_2,\ldots,v_k},$$

et soient $\zeta_1, \zeta_2, \ldots, \zeta_l$ des cellules. Alors la notation

$$\theta(\zeta_1, \zeta_2, \ldots, \xi_l)$$

s'appelle fonction de l variables ou *fonction* (*opération*) à *l éléments* et désigne une cellule sans nom dont l'état est le résultat de l'opération θ appliquée aux états des cellules $\zeta_1, \zeta_2, \ldots, \zeta_l$; θ n'est pas formalisé dans le YALS *).

Remarquons que chaque composition de fonctions est encore une fonction dans le YALS, de sorte que, par exemple, la notation

$$\theta_1(\theta_2(\zeta_1, \zeta_2), \theta_3(\zeta_1, \zeta_3), \zeta_3)$$

que l'on peut également désigner par

$$\theta_4(\zeta_1, \zeta_2, \zeta_3)$$

exprime une cellule sans nom dont l'état est le résultat de l'opération en question.

Les cellules $\zeta_1, \zeta_2, \ldots, \zeta_l$ figurant dans la notation $\theta(\zeta_1, \zeta_2, \ldots, \zeta_l)$ sont appelées *arguments* de la fonction θ. On dit de l'argument ζ_i qu'il est *non essentiel* si pour toute combinaison des états des cellules $\zeta_1, \zeta_2, \ldots, \zeta_l$ pour laquelle la fonction à l éléments θ est définie, toute variation admissible de l'état de la cellule ζ_i ne change pas le résultat de l'opération θ (dans le cas contraire ζ_i est un argument *essentiel* de θ).

Il est clair que si ζ_i est un argument non essentiel d'une fonction à l éléments, il existe une fonction θ' à $l - 1$ éléments identiquement égale à θ et n'ayant pas d'argument ζ_i.

Exemple 2.1. On a vu des arguments non essentiels dans l'analyse mathématique. Par exemple, pour la fonction $f(x, y) = x^2 + \sin^2 y + \cos^2 y$, y est un argument non essentiel et x un argument essentiel. Elle est identiquement égale à la fonction d'une variable $\varphi(x) = x^2 + 1$.

Toute cellule qui n'est pas argument d'une fonction dans le YALS peut être considérée comme un argument non essentiel de cette fonction. Par exemple, on peut dire que sin x est une fonction de deux variables $\| f_1(x, y) \|$.

On a déjà dit que, pour certaines cellules (à savoir, pour les paramètres), la notion d'« état de cellule » est définie dans le YALS. Pour désigner des opérations sur les valeurs des paramètres, on

*) C'est-à-dire que le symbole θ doit être défini pour chaque application concrète du YALS.

emploie les signes usuels d'opérations mathématiques cités dans la liste des symboles du YALS. Pourtant, chaque fois qu'on est amené à considérer des opérations n'ayant pas de désignations usitées, on est libre d'employer les signes d'opérations non formalisées.

Une fonction dont les valeurs sont les états de la cellule logique s'appelle *prédicat*. Les prédicats non formalisés sont désignés par la lettre p avec ou sans indices.

Un cas spécial des prédicats, qu'on appelle relations, est aussi formalisé dans le YALS. On sous-entend par *relation* un couple de fonctions de paramètres reliées par un signe de relation (i.e. par l'un des signes $\| > \| \geqslant \| < \| \leqslant \| = \| \neq \|$). Les valeurs d'une relation sont les états de la cellule logique : 1 (**vrai**) si la relation est satisfaite et 0 (**faux**) dans le cas contraire.

EXEMPLE 2.2. Voici des exemples de relations :

$$\| j_1 < 2 \| \; i + j_2 = 10 \| \; i^2 + j^2 > k^2 \|.$$

On peut construire, moyennant les connecteurs logiques, des prédicats plus compliqués, par exemple :

$$\| j_1 < 2 \wedge i^2 + j^2 > k^2 \vee i + j_2 = 10 \|.$$

Le dernier prédicat peut être désigné comme suit :

$$p_2 (i, j, k, j_1, j_2) \|.$$

Lorsque les objets sont des grandeurs mathématiques, on peut construire le prédicat

$$\| i > 2 \wedge x_i > 3{,}5 \|,$$

ce qu'on peut désigner comme

$$\| p_1 (i, x_i) \|.$$

En définissant, pour une application concrète du YALS, les états d'objets, on peut introduire dans le YALS de nouveaux symboles de relation. Par exemple, si les objets sont des ensembles et leurs éléments, on peut utiliser les signes de relations $\subset$ (est inclus dans) et $\in$ (est un élément de) ; ainsi, on peut imaginer un prédicat de la forme

$$\| x \subset y \wedge z \in x \|.$$

Remarquons qu'une fonction identique $\theta (\zeta)$ qui vérifie la condition

$$\theta (\zeta) \equiv \zeta$$

peut être notée, si l'on veut, tout simplement comme ζ.

Signes de passage. Les constructions formées avec les symboles

de base et ayant la forme $\|\ulcorner^{v_1}\|\llcorner_{v_2}\|\lrcorner_{v_3}\|$, où v_1, v_2, v_3 sont des nombres naturels, s'appellent *signes de passage*.

De même que le demi-crochet qui en fait partie, le signe $\ulcorner^{v_1}$ est dit *ouvrant supérieur*, le signe $\llcorner_{v_2}$ *ouvrant inférieur*, et le signe $\lrcorner_{v_3}$, *fermant*. On voit qu'il existe un seul signe de passage fermant, mais deux signes de passage ouvrants. La signification qu'on attribue aux signes de passage sera expliquée plus bas.

Cellules, fonctions et signes de passage ouvrants dépendant de paramètres. Lorsqu'on remplace, dans la notation d'une cellule, d'une fonction ou d'un signe de passage ouvrant, les indices numériques (éventuellement quelques-uns) par des fonctions à valeurs entières de paramètres, on obtient respectivement:

une *cellule dépendant de paramètres*,
une *fonction dépendant de paramètres* et
un *signe de passage ouvrant dépendant de paramètres.*

Dans ce qui suit, nous entendons par cellules, fonctions et signes de passage ouvrants respectivement les cellules, les fonctions et les signes de passage ouvrants dépendant de paramètres. Les cellules, les fonctions et les signes de passage ouvrants seront considérés comme des cas spéciaux de cellules, fonctions et signes de passage ouvrants dépendant de paramètres, mais conservant leurs valeurs indépendamment de celles de paramètres.

2.1.3. Langage des opérandes lié au YALS. Le langage des opérandes lié au YALS sert à la description des états de la mémoire dont on a parlé plus haut. Son alphabet est composé des symboles de base suivants:

— les lettres latines minuscules italiques, sauf *f* et *p*;
— les chiffres décimaux $\|0\|1\|2\|3\|4\|5\|6\|7\|8\|9\|$;
— les valeurs logiques **vrai** et **faux** désignées respectivement par $\|1\|$ et $\|0\|$ (il faut nettement distinguer les cas où 1 et 0 sont des chiffres et où ils sont des valeurs logiques);
— le signe d'égalité $\|=\|$ qui dans le langage des opérandes se lit comme « est en état », et le séparateur $\|;\|$ (point-virgule).

Les morphèmes du langage des opérandes sont les nombres entiers et les cellules. Ces structures primaires sont décrites au p. 2.1.2.

La syntaxe du langage des opérandes est représentée par deux règles et deux conventions.

1) Règle de description de l'état d'une cellule. La description de l'état d'une cellule se compose d'un nom de cellule suivi du signe d'égalité, suivi de l'état de cellule.

2) Règle de description de l'état de mémoire. Une description de l'état de mémoire est une ligne qui

représente une suite de descriptions d'états de cellules séparées par les signes || ; || (point-virgule).

3) Première convention. Si l'état d'une cellule est constant, on peut choisir cet état pour désigner la cellule. La même convention est admise dans le YALS.

4) Deuxième convention. Dans une description de l'état de mémoire qui contient une description de l'état d'une cellule de la forme $\zeta = \zeta$, on peut barrer cette dernière description. A toute description de l'état de mémoire qui ne contient pas de description de l'état de cellule de la forme

$$\zeta = \zeta,$$

on peut ajouter une description de cette forme.

EXEMPLE 2.3. L'état d'une cellule paramètre peut être décrit comme
$\| j_1 = 123; \|$ ou, ce qui est le même, comme $\| j \downarrow 1 \uparrow = 123; \|$. Les deux notations signifient « cellule j_1 est en état 123 ».

Lorsqu'on applique le YALS à la description d'une procédure de calcul, l'état d'une cellule objet peut être décrit comme suit:

$$\| x = -21{,}0395; \|.$$

Si les objets sont des textes de la langue française la description suivante de l'état de cellule est possible:

$$\| x_2 = \text{dans le cas suivant}; \|.$$

2.1.4. Opérateurs élémentaires. On appelle *opérateur élémentaire principal* une notation de la forme

$$\| \zeta : = \theta (\zeta_1, \zeta_2, \ldots, \zeta_k); \|,$$

où $\zeta_1, \zeta_2, \ldots, \zeta_k$ sont des cellules (indépendantes de paramètres), et θ une fonction (indépendante de paramètres). Le cas où ζ coïncide avec l'une des cellules $\zeta_1, \zeta_2, \ldots, \zeta_k$ est admissible. Lorsque dans un opérateur élémentaire principal l'une au moins des cellules ou le signe de fonction sont respectivement cellule ou signe de fonction dépendant de paramètres, la notation obtenue s'appelle *opérateur élémentaire principal dépendant de paramètres*. Pour abréger, nous omettons partout dans la suite le mot « principal ».

En se donnant des valeurs déterminées des paramètres, on passe d'un opérateur élémentaire dépendant de paramètres à sa réalisation particulière (à un opérateur élémentaire qui ne dépend plus de paramètres). Un opérateur élémentaire qui ne dépend pas de paramètres est considéré comme un cas spécial d'opérateur dépendant de paramètres.

On appelle exécution d'un opérateur élémentaire la détermination de sa réalisation suivie d'un changement d'état de mémoire.

Ce procédé consiste en ce qu'on détermine la valeur de la fonction d'après les états des cellules figurant dans la partie droite, cette valeur étant dorénavant considérée comme l'état de la cellule écrite dans la partie gauche de la réalisation de l'opérateur élémentaire.

En plus des opérateurs principaux, il existe les *opérateurs élémentaires de passage*. Ce sont les signes de passage qu'on a déjà décrits, ouvrants ou fermants, dépendant ou non de paramètres.

Dans ce qui suit les opérateurs élémentaires de passage seront appelés, comme avant, signes de passage.

2.1.5. Opérateurs; classes d'opérateurs dans le YALS. On appelle *opérateur* une suite d'opérateurs élémentaires, où chaque opérateur est suivi du symbole || ;|| (point-virgule), cette suite peut être disposée en ligne ou en colonne.

On appelle exécution d'un opérateur le processus d'exécution successive de tous ses opérateurs élémentaires. Ainsi, un opérateur représente la description d'une transformation de la mémoire.

Dans le langage des schémas logiques on distingue les classes suivantes d'opérateurs:

1. *Opérateurs d'action*, i.e. les opérateurs dont toutes les cellules sont des objets.

2. *Opérateurs de variation*, i.e. les opérateurs dont toutes les cellules sont des paramètres.

3. *Opérateurs logiques*, i.e. les opérateurs élémentaires où les fonctions sont des prédicats (d'objets et de paramètres) et dont la partie gauche est la cellule logique o. Comme la notation de tout opérateur logique commence par les symboles $\| o := \|$, on convient, pour abréger les notations, d'omettre ces symboles, en n'écrivant que le prédicat (suivi du symbole || ;||).

4. *Opérateurs de formation*, i.e. les opérateurs dont les arguments de toutes les fonctions sont des objets, et dont les parties gauches des opérateurs élémentaires sont des paramètres.

5. *Opérateurs d'entrée d'objets* (appelés également *opérateurs d'appel d'objets*), i.e. les opérateurs dont tous les arguments de fonctions sont des paramètres et toutes les cellules dans les parties gauches des opérateurs élémentaires sont des objets.

En plus des opérateurs cités, on utilise dans le YALS les symboles I_0 et T_ν (ν est un nombre naturel) qu'on appelle également opérateurs (I_0 est l'*opérateur initial*, T_ν l'*opérateur terminal* d'un processus).

Un opérateur dont les réalisations peuvent être distinctes l'une de l'autre s'appelle opérateur dépendant de paramètres.

On obtient une réalisation particulière d'un opérateur qui comprend des opérateurs élémentaires dépendant de paramètres en remplaçant lesdits opérateurs élémentaires par leurs réalisations. Au p. 2.1.7. nous décrirons une autre méthode d'obtention de réalisa-

tions d'opérateurs. Les opérateurs ne dépendant pas de paramètres sont considérés comme un cas spécial d'opérateurs dépendant de paramètres.

EXEMPLE 2.4. L'opérateur dépendant de paramètres

$$\| x: = f_i (x_i, y_j); \; y_i := f (x_{i+1}, y_{y-1}, u); \|$$

a, pour $i = 2$, $j = 5$, la réalisation

$$\| x: = f_2 (x_2, y_5); \; y_2: = f (x_3, y_4, u); \|.$$

C'est un opérateur d'action.

L'opérateur logique

$$\| x \leqslant 10 \supset j < 5; \|$$

ne dépend pas de paramètres, bien qu'il contienne un prédicat de j.

Au contraire, l'opérateur

$$\| x_j < 25 \vee i > 6; \|$$

dépend du paramètre j (mais non pas de i).

Exécuter un opérateur c'est obtenir sa réalisation et exécuter successivement tous les opérateurs élémentaires de cette réalisation.

2.1.6. Signes d'opérateurs et d'opérateurs dépendant de paramètres. Le YALS possède des moyens d'abréger les désignations d'opérateurs. Soient ν et $\mu_1, \mu_2, \ldots, \mu_k$ des nombres naturels, et Z l'une des lettres D, V, P, F, E, désignant les types d'opérateurs: d'action, de variation, logiques, de formation et d'entrée d'objets.

En tant que nom d'opérateurs on emploie l'un des symboles

$$\| Z_\nu \| Z_\nu^{\mu_1, \mu_2, \ldots, \mu_k} \|.$$

Si l'on remplace, dans cette notation, l'un au moins des indices supérieurs par une fonction à valeurs entières de quelques paramètres, on obtient la notation utilisée pour désigner les opérateurs dépendant des paramètres qui sont arguments de la fonction en question. Au lieu d'une notation contenant des indices supérieurs ou inférieurs, on peut se servir de la notation correspondante avec les symboles $\uparrow$ et $\downarrow$. On aura respectivement

$$\| Z \downarrow \nu \uparrow \| Z \uparrow \mu_1, \mu_2, \ldots, \mu_k \downarrow \downarrow \nu \uparrow \|.$$

La dernière notation peut être remplacée par la notation équivalente

$$\| Z \downarrow \nu \uparrow \uparrow \mu_1, \mu_2, \ldots, \mu_k \downarrow \|.$$

En choisissant pour désigner un opérateur dépendant de paramètres une des notations proposées, on obtient ses réalisations en

concrétisant la désignation adoptée. Par exemple, la réalisation de l'opérateur $Z_8^{i,j}$ pour $i = 3$, $j = 4$ sera désignée comme $Z_8^{3,4}$.

2.1.7. Notation d'un algorithme en YALS. Schéma logique et décodage des opérateurs. On peut noter un algorithme en YALS à l'un des trois niveaux : abstrait, mixte et concret. Donnons d'abord les règles de notation d'un algorithme au niveau abstrait. A ce niveau, la notation d'un algorithme se compose de deux parties qui sont le schéma logique de l'algorithme et son décodage.

On appelle *expressions élémentaires*:

a) une notation de la forme $I_0 \underset{\mu}{\llcorner}$;

b) l'opérateur terminal T_μ;

c) une notation de la forme $Q \underset{\nu}{\llcorner}$, où Q est le symbole d'un opérateur non logique distinct des opérateurs initial et terminal;

d) des notations de la forme $P \underset{\nu_1}{\llcorner} \overset{\nu_2}{\ulcorner}$ ou $P \overset{\nu_2}{\ulcorner} \underset{\nu_1}{\llcorner}$, où P est le symbole d'opérateur logique;

e) le signe de passage fermant $\underset{\mu}{\lrcorner}$.

Il n'existe pas d'autres expressions élémentaires. Ici μ est un entier, et ν, ν_1, ν_2 sont des fonctions à valeurs entières de paramètres ou des nombres entiers.

On appelle *schéma logique* une suite finie d'expressions élémentaires commençant par l'expression élémentaire $I_0 \underset{\mu}{\llcorner}$ qu'on ne rencontre plus dans la suite.

Le dernier symbole d'un schéma logique doit être suivi d'un crochet ouvrant, ensuite vient le *décodage* du schéma logique et enfin un crochet fermant. Le décodage d'un schéma logique représente une suite d'expressions obtenues par décodage de tous ses opérateurs, sauf I_0 et T_ν.

Si un opérateur quelconque se rencontre dans un schéma logique plusieurs fois, son décodage ne doit figurer qu'une seule fois.

Le décodage d'un opérateur est une suite composée d'un symbole d'opérateur et d'une notation représentant cet opérateur comme une succession d'opérateurs élémentaires (s'il n'est pas logique) ou comme un prédicat (s'il est logique). La notation doit être conforme aux règles formulées plus haut.

EXEMPLE 2.5. Un algorithme en YALS, équivalent à l'algorithme d'Euclide (exemple 1.27) est:

$$\| I_0 \underset{1}{\llcorner} \; \underset{1}{\lrcorner} P_1 \overset{2}{\ulcorner} \underset{3}{\llcorner} \; \underset{3}{\lrcorner} P_2 \overset{4}{\ulcorner} \underset{5}{\llcorner} \; \underset{5}{\lrcorner} D_1 \underset{6}{\llcorner} \; \underset{6}{\lrcorner} T_1 \; \underset{2}{\lrcorner} D_2 \; \underset{1}{\llcorner} \; \underset{4}{\lrcorner} D_3 \underset{1}{\llcorner} \; [P_1 x > y;$$

$$P_2 y > x; \; D_1 z := x; \; D_2 x := x - y; \; D_3 y := y - x;] \|.$$

Nous avons dit « équivalent à l'algorithme d'Euclide » et non « algorithme d'Euclide », puisque *primo* il est donné dans un langage algorithmique différent de celui de l'exemple 1.27, et *secundo* son algorithme d'exécution est différent. Par la suite nous appellerons algorithme d'Euclide cet algorithme équivalent.

Par x, y, z sont désignées les variables entières. A gauche du symbole $\| [\|$ on a le schéma logique, et à sa droite (entre les symboles $\| [\| \;] \|$) on a le décodage des opérateurs.

Si l'on remplace, dans le schéma logique d'un algorithme, certains (mais pas tous) symboles d'opérateurs par leurs notations complètes, prises dans le décodage, et que l'on supprime, dans le décodage du schéma logique, les décodages des opérateurs dont les symboles ne figurent plus à gauche du crochet ouvrant, on obtiendra ce qu'on appelle notation de l'algorithme au niveau mixte du YALS.

Exemple 2.6. En remplaçant dans l'algorithme de l'exemple 2.5 P_1 et D_2 par leurs décodages, on obtient une notation au niveau mixte

$$\| I_0 \underset{1}{\llcorner} \; \underset{1}{\lrcorner} x > y ; \overset{2}{\ulcorner} \underset{3}{\llcorner} \; \underset{3}{\lrcorner} P_2 \overset{4}{\ulcorner} \underset{5}{\llcorner} \; \underset{5}{\lrcorner} D_1 \underset{6}{\llcorner} \; \underset{6}{\lrcorner} . T_1 \underset{2}{\lrcorner} x :=$$

$$x - y ; \underset{1}{\llcorner} \; \underset{4}{\lrcorner} D_3 \underset{1}{\llcorner} [P_2 y > x ; \; D_1 z := x ; \; D_3 y = y - x ;] \| .$$

Si dans le schéma logique on remplace tous les symboles d'opérateurs par leurs notations développées en épuisant la partie décodage, et que l'on supprime ensuite les crochets, on obtient un algorithme *au niveau concret* du langage des schémas logiques. On peut dire que, sous cette forme un schéma logique et son décodage sont identiques.

Exemple 2.7. L'algorithme d'Euclide au niveau concret a la forme

$$\| I_0 \underset{1}{\llcorner} \; \underset{1}{\lrcorner} x > y ; \overset{2}{\ulcorner} \underset{3}{\llcorner} \; \underset{3}{\lrcorner} y > x ; \overset{4}{\ulcorner} \underset{5}{\llcorner} \; \underset{5}{\lrcorner} z := x ;$$

$$\underset{6}{\llcorner} \; \underset{6}{\lrcorner} T_1 \underset{2}{\lrcorner} x := x - y ; \underset{1}{\llcorner} \; \underset{4}{\lrcorner} y := y - x ; \underset{1}{\llcorner} \| .$$

Un cas spécial de définition d'opérateurs dépendant de paramètres. Nous avons déjà mentionné (p. 2.1.5) une possibilité de définir en YALS des opérateurs dépendant de paramètres ; il s'agit d'utiliser des symboles désignant ces opérateurs et d'écrire dans la partie décodage non pas le décodage de l'opérateur, mais le décodage de

ses réalisations qui peuvent se rencontrer lors de l'exécution de l'algorithme.

EXEMPLE 2.8. Voici un algorithme où l'opérateur D_2^j est donné par la méthode décrite :

$$\| I_0 \llcorner_1 \; \lrcorner_1 D_1 \llcorner_2 \; \lrcorner_2 V_1 \llcorner_3 \; \lrcorner_3 V_2 \llcorner_4 \; \lrcorner_4 D_2^j \llcorner_5 \; \lrcorner_5 P_1 \llcorner_3 \overset{6}{\ulcorner} \; \lrcorner_6 T_1 [D_1 u := x + y;$$

$$V_1 j := 0; V_2 j := j + 1; \quad D_2^1 v := u \cdot x; \; D_2^2 v := v \cdot y;$$

$$D_2^3 v := v - x; \; P_1 j = 3;] \|.$$

Cet algorithme calcule, d'après les valeurs de x et y, les valeurs des quantités

$$u = x + y,$$

$$v = (x + y) \cdot x \cdot y - x.$$

Ce mode de description est conseillé lorsque l'opérateur dépendant de paramètres a un nombre réduit de réalisations ; par contre, on ne demande pas que toutes les réalisations soient d'un même type, il suffit que toutes les realisations soient ou bien non logiques, ou bien logiques.

2.1.8. Règles d'exécution d'un algorithme donné dans le YALS. Le type de réalisation d'opérateurs (d'action, de variation, logique, de formation, d'entrée d'objets) sera désigné respectivement par D, V, P, F, E. Les mots « est du type » seront exprimés par le signe d'égalité.

Supposons qu'avant l'exécution de l'algorithme la mémoire des objets se trouve en état δ, celle des paramètres, en état ε.

La règle d'exécution d'algorithmes donnés en YALS (pour fixer les idées, nous considérons la notation au niveau abstrait) peut être formulée de la manière suivante.

1°. Poser $\varepsilon_0 = \varepsilon$, $\delta_0 = \delta$, $\omega_0 = 0$, $\rho = 0$. Trouver dans le schéma logique l'opérateur initial et aborder le p. 2°.

2°. Passer à l'élément suivant du schéma logique. Aborder le p. 3°.

3°. Si le symbole considéré est l'opérateur terminal T_ν arrêter le processus ; sinon (c'est-à-dire, dans tous les autres cas) aborder le p. 4°.

4°. Si le symbole considéré est un signe de passage fermant, revenir au p. 2° ; sinon aborder le p. 5°.

5°. Si le symbole considéré n'est pas un opérateur (autre que I_0 et T_1), aborder le p. 6° ; sinon déterminer sa réalisation K qui

correspond à l'état ε_ρ de la mémoire des paramètres. Poser

$$\delta_{\rho+1} = \begin{cases} D(\delta_\rho) \text{ si } K = D; \\ E(\delta_\rho, \varepsilon_\rho) \text{ si } K = F; \\ \delta_\rho \text{ dans les autres cas}; \end{cases}$$

$$\varepsilon_{\rho+1} = \begin{cases} V(\varepsilon_\rho) \text{ si } K = V; \\ F(\varepsilon_\rho, \delta_\rho) \text{ si } K = F; \\ \varepsilon_\rho \text{ dans les autres cas}; \end{cases}$$

$$\omega_{\rho+1} = \begin{cases} P(\delta_\rho, \varepsilon_\rho) \text{ si } K = P; \\ \omega_\rho \text{ dans les autres cas.} \end{cases}$$

Autrement dit, exécuter la réalisation K par la méthode déjà décrite. Ensuite, incrémenter ρ d'une unité et revenir au p. 2°.

6°. Si le symbole considéré est un signe de passage ouvrant qui n'est pas immédiatement précédé d'un opérateur non logique, passer au p. 8°; sinon déterminer sa réalisation pour l'état actuel de la mémoire des paramètres. Soit μ l'indice de cette réalisation. Exécuter le p. 7°.

7°. Trouver dans le schéma logique le signe de passage fermant d'indice μ. Revenir au point 2°.

8°. Le symbole considéré et le symbole suivant sont des signes de passage ouvrants. Déterminer leurs réalisations. Choisir pour μ l'indice de la réalisation du signe de passage ouvrant inférieur si $\omega_\rho = 0$, et celui du signe de passage ouvrant supérieur si $\omega_\rho = 1$. Revenir au p. 7°.

Avant de formuler la règle d'exécution d'un algorithme donné en YALS, nous avons fixé les états initiaux des mémoires des objets et des paramètres. Les états de certaines cellules sont essentiels pour l'exécution de l'algorithme, les états des autres ne le sont pas. En disant que les états des mémoires des objets et des paramètres sont donnés, nous voulons dire en réalité que sont donnés les états des cellules essentielles pour l'exécution de l'algorithme.

Remarquons que les états initiaux δ et ε déterminent le déroulement du processus de calcul; il peut: a) se terminer en vertu du p. 3° (après un nombre fini de pas); on dit alors que l'algorithme est *applicable* à δ pour ε; b) tourner indéfiniment; c) s'interrompre sans résultat si l'un des points se trouve inexécutable. Dans les deux derniers cas on dit que l'algorithme *n'est pas applicable* à δ pour ε. Il est également des algorithmes en YALS dont l'exécution s'avère impossible ou infinie à cause des particularités des algorithmes mêmes.

Ceci peut avoir lieu, par exemple, lorsque dans le schéma logique:

— il n'y a pas d'opérateur T_v;

— il y a plusieurs signes de passage fermants de même indice;

— pour un signe de passage ouvrant d'indice numérique (en particulier, dans le cas d'une réalisation dépendant de paramètres), il n'existe pas de signe de passage fermant de même indice.

— le dernier symbole de l'algorithme est un signe de passage fermant.

2.1.9. Expressions. Fermetures des opérateurs. Opérateurs généralisés. Toute suite d'expressions élémentaires, contenant éventuellement l'opérateur I_0, mais seulement à la première place, s'appelle *expression*. Si une telle suite est donnée au niveau abstrait, alors, tout comme une construction logique, elle doit être munie d'un décodage. Les expressions en YALS sont notamment les expressions élémentaires et les algorithmes donnés dans ce langage.

Un cas particulier des expressions est représenté par ce qu'on appelle fermetures des opérateurs. Soit U une expression élémentaire faisant partie d'un algorithme et contenant un opérateur S. En partant de cette expression élémentaire vers la gauche, choisissons tous les signes de passage fermants qui n'en sont pas séparés par des opérateurs. L'ensemble formé par tous ces signes de passage et par l'expression élémentaire elle-même

$$\| \underset{\mu_1}{\lrcorner} \underset{\mu_2}{\lrcorner} \dots \underset{\mu_k}{\lrcorner} U \|$$

s'appelle *fermeture de l'opérateur* S. Seule la fermeture de l'opérateur I_0 ne contient pas de signes de passage fermants, et seules les fermetures des opérateurs T_ν ne contiennent pas de signes de passage ouvrants.

Il est évident que chaque algorithme représente une suite de fermetures d'opérateurs (éventuellement avec des décodages).

Considérons une expression W qui est une suite de fermetures d'opérateurs sans l'opérateur I_0, et qui fait partie d'un algorithme donné en YALS. Répartissons tous les signes de passage fermants en signes intérieurs et signes extérieurs de la manière suivante. Si, dans la notation de l'algorithme et en dehors de W, il y a un signe de passage ouvrant qui admet une réalisation de même indice que le signe de passage fermant donné, alors ce dernier est dit *extérieur*. Dans le cas contraire il est dit *intérieur*. De même, partageons en extérieurs et intérieurs tous les signes de passage ouvrants qui figurent dans l'expression W: si, à toute réalisation d'un signe de passage ouvrant il correspond, dans la même ligne, un signe de passage fermant, alors le signe de passage ouvrant est dit *intérieur* (et *extérieur* dans le cas contraire).

L'opérateur qui est le plus proche à droite du signe de passage fermant extérieur sera appelé *entrée* dans l'expression W, et l'opérateur auquel appartient le signe de passage ouvrant extérieur sera appelé *sortie*.

Une expression W, qui fait partie de la notation d'un algorithme, possède l'unique entrée et ne contient pas d'opérateurs terminaux, peut être déclarée opérateur généralisé et désignée par un seul symbole de la forme $\| R_v \|$. On fait précéder ce symbole par la suite, mise en parenthèses, des signes de passage fermants extérieurs faisant partie de l'expression W, et on le fait suivre de la suite, mise en parenthèses, des signes de passage ouvrants extérieurs de l'expression W, l'ordre de ces signes étant arbitraire. Ainsi, l'expression W se trouve notée dans l'algorithme comme suit:

$$\|(\underset{\mu_1}{\lrcorner}\,\underset{\mu_2}{\lrcorner}\dots\underset{\mu_k}{\lrcorner})\,R_v(L)\|,$$

où L désigne l'ensemble des signes de passage ouvrants extérieurs. Ayant opéré une telle substitution, on inclut dans le décodage du schéma logique le décodage de cette notation.

EXEMPLE 2.9. Dans le schéma logique de l'algorithme

$$\|I_0\underset{1}{\llcorner}\;\underset{1}{\lrcorner}D_1\underset{2}{\llcorner}\;\underset{2}{\lrcorner}P_1\overset{3}{\ulcorner}\underset{4}{\llcorner}\;\underset{4}{\lrcorner}D_2\underset{5}{\llcorner}\;\underset{5}{\lrcorner}P_2\overset{6}{\ulcorner}\underset{7}{\llcorner}\;\underset{7}{\lrcorner}D_3\underset{8}{\llcorner}\;\underset{3}{\lrcorner}\,\underset{8}{\lrcorner}P_3\underset{2}{\llcorner}\overset{9}{\ulcorner}\;\underset{9}{\lrcorner}$$
$$D_4\underset{10}{\llcorner}\;\underset{10}{\lrcorner}P_4\underset{11}{\llcorner}\overset{2}{\ulcorner}\;\underset{6}{\lrcorner}\,\underset{11}{\lrcorner}T_1\ [\text{décodage des opérateurs}]\ \|,$$

l'expression qui commence par le signe $\underset{2}{\lrcorner}$ et finit par $\overset{9}{\ulcorner}$ a une entrée P_1, deux sorties P_2 et P_3 et ne contient pas d'opérateurs terminaux. On peut la considérer comme opérateur généralisé. On aura

$$\|I_0\underset{1}{\llcorner}\;\underset{1}{\lrcorner}D_1\underset{2}{\llcorner}\;(\underset{2}{\lrcorner})R_1(\overset{6}{\ulcorner}\,\overset{9}{\ulcorner})\;\underset{9}{\lrcorner}D_4\underset{10}{\llcorner}\;\underset{10}{\lrcorner}P_4\underset{11}{\llcorner}\overset{2}{\ulcorner}\;\underset{6}{\lrcorner}$$
$$\underset{11}{\lrcorner}T_1\ [R_1\;\underset{2}{\lrcorner}P_1\overset{3}{\ulcorner}\underset{4}{\llcorner}\;\underset{4}{\lrcorner}D_2\underset{5}{\llcorner}\;\underset{5}{\lrcorner}P_2\overset{6}{\ulcorner}\underset{7}{\llcorner}\;\underset{7}{\lrcorner}D_3\underset{8}{\llcorner}\;\underset{8}{\lrcorner}$$
$$\underset{3}{\lrcorner}P_3\underset{2}{\llcorner}\overset{9}{\ulcorner}\ \text{décodage des opérateurs}]\,\|.$$

En remplaçant, dans un algorithme, les opérateurs ou les opérateurs généralisés par leurs symboles littéraux, on simplifie la notation au prix d'une complication de l'expression entre crochets. L'utilisation des expressions littérales a l'avantage de présenter les algorithmes de façon plus suggestive.

2.1.10. Conventions simplifiant la notation d'algorithmes. Convenons avant tout d'écrire, à la place de deux signes de passage ouvrants dont l'un est supérieur et l'autre inférieur, un seul signe de passage ouvrant « double », i.e. posons

$$\|\underset{\alpha}{\llcorner}\overset{\beta}{\ulcorner}=\overset{\beta}{\ulcorner}\underset{\alpha}{\llcorner}=\overset{\beta}{\underset{\alpha}{\sqsubset}}\|.$$

C'est bien légitime, car l'exécution d'un algorithme ne dépend pas de l'ordre de notation de deux signes successifs de passage ouvrants, supérieur et inférieur (voir le p. 8° du paragraphe 2.1.8).

Soit λ_β un signe de passage ouvrant (supérieur ou inférieur). Pour abréger les notations, convenons que

$$\| R\lambda_\beta \underset{\beta}{\lrcorner} = R \underset{\beta}{\lrcorner} \|,$$

où R est ou bien Q, ou bien $P\underset{\alpha}{\llcorner}$, ou enfin $P\overset{\alpha}{\ulcorner}$; α, β = const. Autrement dit, convenons que

$$\| Q \underset{\beta}{\llcorner} \underset{\beta}{\lrcorner} = Q \underset{\beta}{\lrcorner} \|,$$

$$\| P \overset{\beta}{\underset{\alpha}{\sqsubset}} \underset{\beta}{\lrcorner} = P \underset{\alpha}{\llcorner} \|,$$

$$\| P \overset{\alpha}{\underset{\beta;}{\sqsubset}} \underset{\beta}{\lrcorner} = P \overset{\alpha}{\ulcorner} \|,$$

où Q est un opérateur quelconque autre que logique (sauf T_v), et P un opérateur logique.

Remarquons qu'un signe de passage fermant « solitaire » $\underset{\beta}{\lrcorner}$ peut être omis ou écrit devant ou derrière n'importe quelle fermeture d'opérateur dans un schéma logique (dans un algorithme).

Nous entendons par « solitaire » un signe de passage fermant dont l'indice ne figure pas parmi les indices des signes de passage ouvrants (ni parmi les réalisations des signes de passage ouvrants dépendant de paramètres) de l'algorithme donné.

Tout à fait de même, remarquons qu'on peut permuter deux signes de passage fermants voisins:

$$\| \underset{\alpha}{\lrcorner} \underset{\beta}{\lrcorner} = \underset{\beta}{\lrcorner} \underset{\alpha}{\lrcorner} \|.$$

De plus, si l'on a côte à côte deux signes de passage fermants, on peut supprimer l'un d'eux, tout en ayant soin de remplacer ceux des indices des signes ouvrants qui sont identiques à celui du signe supprimé par les indices du signe fermant qui reste (cette règle n'est pas applicable au cas où l'indice d'une réalisation d'un signe de passage ouvrant est identique à celui du signe de passage fermant que nous voulons supprimer).

Exemple 2.10. Voici la notation d'un algorithme avant la simplification :

$$\| I_0 \underset{1}{\llcorner} \underset{1}{\lrcorner} D_1 \underset{2}{\llcorner} \underset{2}{\lrcorner} P_1 \overset{3}{\ulcorner} \underset{4}{\llcorner} \underset{4}{\lrcorner} D_2 \underset{5}{\llcorner} \underset{5}{\lrcorner} P_2 \overset{6}{\ulcorner} \underset{7}{\llcorner} \underset{7}{\lrcorner} D_3$$

$$\underset{8}{\llcorner} \underset{8}{\lrcorner} \underset{3}{\lrcorner} P_3 \underset{2}{\llcorner} \overset{9}{\ulcorner} \underset{9}{\lrcorner} D_4 \underset{10}{\llcorner} \underset{6}{\lrcorner} \underset{11}{\lrcorner} T_1 \underset{10}{\lrcorner} P_4 \underset{11}{\llcorner} \overset{2}{\ulcorner} \text{[décodage des opérateurs]} \|.$$

Après la simplification on a:

$$\| I_0 D_1 \underset{2}{\lrcorner} P_1 \overset{3}{\ulcorner} D_2 P_2 \overset{6}{\ulcorner} D_3 \underset{3}{\lrcorner} P_3 \underset{2}{\llcorner} D_4 \underset{10}{\llcorner} \underset{6}{\lrcorner} T_1 \underset{10}{\lrcorner} P_4 \overset{2}{\underset{6}{\sqsubset}}$$

[décodage des opérateurs] ||.

Pour ne pas mentionner, chaque fois qu'on étudie un algorithme, si les conventions formulées plus haut sont utilisées ou non, nous convenons que les signes de passage ouvrants et les signes de passage fermants qui leur correspondent sont *toujours* présents dans nos notations, d'une manière *explicite* ou *implicite*.

2.1.11. Réunion d'algorithmes. En pratique, on utilise parfois des notations qui réunissent plusieurs algorithmes. Voici l'une des méthodes de réunion. Au lieu d'un seul symbole I_0, on emploie un nombre arbitraire (nécessaire) de symboles I_1, I_2, ... Ces symboles (tout comme I_0) sont appelés opérateurs initiaux. On convient que, dans la notation d'un algorithme, un opérateur initial peut occuper la première place, ou bien suivre un couple de symboles $Q \underset{\alpha}{\llcorner}$, où Q est un opérateur autre que logique mais différent de T_ν, ou bien suivre un couple de symboles $P \overset{\beta}{\underset{\alpha}{\sqsubset}}$, où P est un opérateur logique, ou enfin suivre un opérateur terminal T_ν.

Dans une réunion d'algorithmes, on peut exécuter l'un d'eux. Pour le repérer, il faut indiquer l'opérateur initial par lequel commence son exécution.

§ 2.2. Langages machine des opérandes

Pour mieux comprendre le fonctionnement du calculateur, il est utile de considérer les opérations élémentaires par lesquelles s'exécutent les opérations générales (voir p. 1.5.6). Il faut donc considérer les langages des opérandes de machine relativement à ces opérations élémentaires.

Dans les calculateurs modernes on utilise des alphabets à deux lettres. Convenons de désigner les lettres employées par || 0 || 1 ||. Un langage des opérandes se compose habituellement de trois sous-langages: un système de représentation des nombres en virgule fixe, un système de représentation des nombres en virgule flottante et un système de représentation de l'information alphanumérique.

2.2.1. Langage des nombres en virgule fixe. Pour décrire la syntaxe du langage de représentation des nombres en virgule fixe, nous allons nous servir de la notation normale de Backus.

⟨lettre⟩ : : = 0 | 1
⟨signe⟩ : : = ⟨+⟩ | ⟨−⟩

$$\langle + \rangle ::= \underbrace{00 \ldots 0}_{k \text{ lettres}}$$

$$\langle - \rangle ::= \underbrace{11 \ldots 1}_{k \text{ lettres}}$$

$$\langle \text{partie entière} \rangle ::= \underbrace{\langle \text{lettre} \rangle \langle \text{lettre} \rangle \cdots \langle \text{lettre} \rangle}_{n \text{ lettres}}$$

$$\langle \text{partie fractionnaire} \rangle ::= \underbrace{\langle \text{lettre} \rangle \langle \text{lettre} \rangle \cdots \langle \text{lettre} \rangle}_{\nu \text{ lettres}}$$

⟨nombre en virgule fixe⟩ : : = ⟨signe⟩ ⟨partie entière⟩ ⟨partie fractionnaire⟩

La partie droite de la dernière formule avec indication des longueurs de ses parties constitutives s'appelle *format* du nombre en virgule fixe et a la forme

$$\underbrace{\langle \text{signe} \rangle}_{k} \underbrace{\langle \text{partie entière} \rangle}_{n} \underbrace{\langle \text{partie fractionnaire} \rangle}_{\nu}$$

Dans bien des calculateurs, le format des nombres en virgule fixe est constant.

Un langage des nombres en virgule fixe muni d'un algorithme établissant sa sémantique s'appelle code des nombres en virgule fixe. En pratique, on fait l'usage des codes droit, inverse et complémentaire.

Ces codes permettent non seulement de représenter les nombres dans les calculateurs, mais aussi d'obtenir les résultats de diverses opérations mathématiques et logiques sur les nombres à l'aide d'opérations simples sur leurs codes.

2.2.2. Additionneurs. Dans les calculateurs utilisant le système de numération binaire les opérations d'addition et de soustraction se font dans des dispositifs spéciaux appelés *additionneurs binaires*. Soient α et β deux entiers positifs à m chiffres binaires:

$$\alpha = a_{m-1}a_{m-2} \ldots a_1a_0,$$

$$\beta = b_{m-1}b_{m-2} \ldots b_1b_0.$$

Ici a_i et b_i sont des chiffres binaires ($i = 0, 1, 2, \ldots, m-1$). Pour chaque $i = 0, 1, \ldots, m-1$, un additionneur binaire à m positions effectue les opérations

$$c_i = a_i + b_i + z_i - 10z_{i+1},$$

où

$$z_{i+1} = \begin{cases} 0 \text{ pour } a_i + b_i + z_i < 10, \\ 1 \text{ pour } a_i + b_i + z_i \geqslant 10. \end{cases} \quad (2.1)$$

Ici z_i est la retenue provenant de la position binaire $(i - 1)$ et dont il faudra tenir compte à la position i, sa valeur est 0 ou 1, c_i est le chiffre du résultat, 10 désigne le nombre 2.

Les formules (2.1) ne définissent pas la quantité z_0. Suivant le mode de définition de cette quantité, on distingue deux types d'additionneurs binaires à m positions:

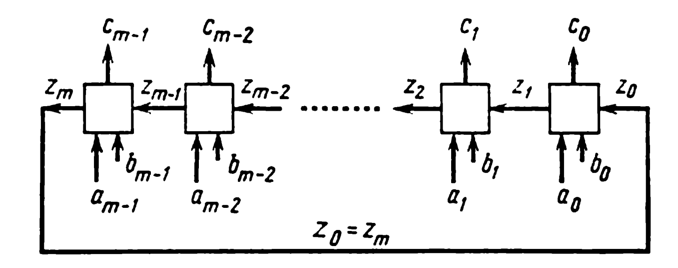


Fig. 2.1. Fonctionnement d'un additionneur avec report cyclique

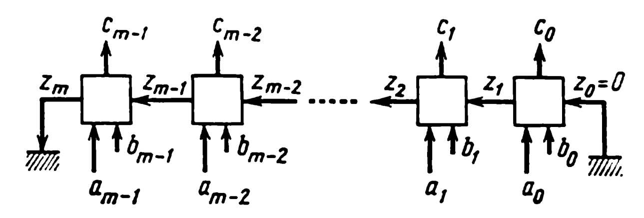


Fig. 2.2. Fonctionnement d'un additionneur sans report cyclique

— les *additionneurs avec transfert cyclique de la retenue de la position* $i = m - 1$ *à la position* $i = 0$ (fig. 2.1), pour lesquels

$$z_0 = z_m;$$

— *les additionneurs sans transfert de la retenue de la position* $i = m - 1$ à la position $i = 0$ (fig. 2.2), pour lesquels

$$z_0 = 0,$$

et la quantité z_m n'est pas utilisée.

Il est clair que le résultat obtenu à l'aide d'un additionneur binaire à m positions ne représente la somme arithmétique des nombres α et β que dans le cas où $z_m = 0$, c'est-à-dire $\alpha + \beta < 10^m$ (10 désigne 2).

L'opération que réalise un additionneur avec report cyclique (désignons-la par le symbole $\underset{m}{\oplus}$) est définie par la formule

$$\alpha \underset{m}{\oplus} \beta = \begin{cases} \alpha+\beta & \text{pour } \alpha+\beta < 10^m, \\ \alpha+\beta-10^m+1 & \text{pour } \alpha+\beta \geqslant 10^m. \end{cases} \quad (2.2)$$

En termes de la théorie des nombres cette opération s'appelle somme modulo $10^m - 1$ et se désigne par $(\alpha + \beta) \bmod (10^m - 1)$.

Un additionneur sans report cyclique effectue l'opération (désignons-la par le symbole $\underset{m}{\boxplus}$) définie par la formule

$$\alpha \underset{m}{\boxplus} \beta = \begin{cases} \alpha+\beta & \text{pour } \alpha+\beta < 10^m, \\ \alpha+\beta-10^m & \text{pour } \alpha+\beta \geqslant 10^m. \end{cases} \quad (2.3)$$

Dans la théorie des nombres, cette opération s'appelle somme modulo 10^m et se désigne par $(\alpha + \beta) \bmod 10^m$.

L'addition et la soustraction de nombres positifs et négatifs ne sont réalisables à l'aide d'additionneurs binaires à m positions que lorsque les nombres sont représentés en codes spéciaux qui doivent, évidemment, satisfaire à la condition:

$$0 \leqslant f(N) < 10^m, \quad (2.4)$$

où N est un nombre et $f(N)$ son code considéré comme un nombre binaire. De plus, pour que le résultat d'une addition soit exprimé dans le même code que les données initiales, les conditions suivantes sont à satisfaire:

1) pour l'additionneur avec report cyclique

$$f(N_1) \underset{m}{\oplus} f(N_2) = f(N_1+N_2); \quad (2.5)$$

2) pour l'additionneur sans report cyclique

$$f(N_1) \underset{m}{\boxplus} f(N_2) = f(N_1+N_2). \quad (2.6)$$

Par N_1 et N_2 sont désignés les nombres à additionner.

2.2.3. Code droit. Soit x un nombre binaire à n chiffres de la partie entière et ν chiffres de la partie fractionnaire:

$$x = sa_{n-1}a_{n-2} \ldots a_0, a_{-1} \ldots a_{-\nu}, \quad (2.7)$$

s étant le signe du nombre et a_i ses chiffres binaires.

On appelle *code droit* $[x]^k_{\text{dr}}$ du nombre x le nombre entier à $n + \nu + k$ chiffres ($k \geqslant 1$) défini par la formule

$$[x]^k_{\text{dr}} = \begin{cases} \underbrace{00\ldots0}_{k \text{ zéros}} a_{n-1}a_{n-2} \ldots a_0a_{-1} \ldots a_{-\nu} & \text{pour } x \geqslant 0, \\ \underbrace{11\ldots1}_{k \text{ unités}} a_{n-1}a_{n-2} \ldots a_0a_{-1} \ldots a_{-\nu} & \text{pour } x \leqslant 0. \end{cases} \quad (2.8)$$

Les k poids forts du code droit représentent le *signe* du nombre.

EXEMPLE 2.11. On remarquera que le nombre zéro a deux représentations possibles en code droit qu'on appelle respectivement zéro positif et zéro négatif:

$$[0]^k_{\mathrm{dr}} = [+0]^k_{\mathrm{dr}} = \underbrace{00\ldots0}_{n+\nu+k \text{ zéros}},$$

$$[0]^k_{\mathrm{dr}} = [-0]^k_{\mathrm{dr}} = \underbrace{11\ldots1}_{k \text{ unités}}\ \underbrace{0\ldots0}_{n+\nu \text{ zéros}}.$$

Pour tout système de codage, la quantité k a une valeur fixe. Afin de déterminer, d'après un code droit, le nombre qui lui correspond, il faut connaître k et ν (n se détermine à partir du nombre de chiffres du code droit).

On utilise le plus souvent en pratique le code droit où le signe est représenté par une position ($k = 1$). Dans la désignation $[x]^k_{\mathrm{dr}}$ avec $k = 1$ nous omettons l'indice supérieur.

EXEMPLE 2.12. Soit $k = 1$, $n = 4$, $\nu = 5$. Alors les codes droits des nombres seront à dix chiffres:

$$[0001,\ 10101]_{\mathrm{dr}} = 0000110101,$$

$$[-0001,\ 10101]_{\mathrm{dr}} = 1000110101.$$

EXEMPLE 2.13. Pour $k = 1$, $n = 0$, $\nu = 9$, les codes droits des nombres seront toujours à dix chiffres:

$$[0{,}001001110]_{\mathrm{dr}} = 0001001110,$$

$$[-0{,}001001110]_{\mathrm{dr}} = 1001001110,$$

$$[-0{,}000110101]_{\mathrm{dr}} = 1000110101.$$

En comparant les dernières égalités de cet exemple et de l'exemple précédent on voit que deux nombres différents peuvent se représenter par des codes droits égaux. Cette éventualité est exclue si l'on considère les nombres ayant les mêmes n, ν et k.

2.2.4. Code inverse. Soit x un nombre binaire à n chiffres dans la partie entière et ν chiffres dans la partie fractionnaire:

$$x = sa_{n-1}a_{n-2}\ldots a_0, a_{-1}\ldots a_{-\nu}.$$

On appelle *code inverse* ou *complément restreint* $[x]^k_{\text{inv}}$ du nombre x le nombre entier à $n+\nu+k$ chiffres ($k \geqslant 1$) défini par la formule

$$[x]^k_{\text{inv}} = \begin{cases} \underbrace{00\ldots 0}_{k \text{ zéros}}\, a_{n-1}a_{n-2}\ldots a_0a_{-1}\ldots a_{-\nu} \text{ pour } x \geqslant 0, \\ \underbrace{11\ldots 1}_{k \text{ unités}}\, \bar{a}_{n-1}\bar{a}_{n-2}\ldots \bar{a}_0\bar{a}_{-1}\ldots \bar{a}_{-\nu} \text{ pour } x \leqslant 0, \end{cases} \qquad (2.9)$$

où

$$\bar{a}_i = 1 - a_i \quad (i = n-1,\ n-2,\ \ldots,\ 0,\ -1,\ \ldots,\ -\nu).$$

Les k poids forts du code inverse représentent le *signe*, les autres éléments, les *chiffres*.

Exemple 2.14. Le nombre zéro admet deux représentations en code inverse qu'on appelle respectivement zéro positif et zéro négatif :

$$[+0]^k_{\text{inv}} = 000\ldots 0 \quad (n+\nu+k \text{ zéros}),$$

$$[-0]^k_{\text{inv}} = 111\ldots 1 \quad (n+\nu+k \text{ unités}).$$

Dans tout système de codage k est fixe.

Lorsque les nombres x et y ont n chiffres dans la partie entière et ν chiffres dans la partie fractionnaire ($m = n+\nu+k$) et

$$|x+y| < 10^n \text{ (en binaire)}, \qquad (2.10)$$

alors les codes inverses de x et y satisfont à (2.4) et (2.5); par conséquent, on peut utiliser un additionneur avec report cyclique pour additionner ces nombres en code inverse.

Si $|x+y| \geqslant 10^n$, la condition (2.5) n'est pas remplie ce qui conduit au dépassement de capacité de l'additionneur.

Exemple 2.15. Supposons que $n = 0$, $\nu = 8$, $k = 2$ et soit $x = 0{,}10001001$, $y = 0{,}11011011$. Alors $x + y = 1{,}01100100 > 10^n = 1$, donc, une addition à report cyclique des codes inverses des nombres x et y conduira à un dépassement de capacité de l'additionneur. En effet, la somme $x+y$ n'a pas de code inverse. Or, l'addition cyclique des nombres $[x]^2_{\text{inv}} = 0010001001$ et $[y]^2_{\text{inv}} = 0011011011$ fournit

$$\begin{array}{r} \oplus\ 0010001001 \\ {}^{10}\ 0011011011 \\ \hline 0101100100 \end{array}$$

Le résultat de l'addition cyclique a, dans les positions du signe, la combinaison 01 et ne représente donc pas un code inverse.

Pour le même x et pour $y = -0{,}01011011$ on aurait $|x + y| = 0{,}00101110 < 10^n = 1$ et

$$\begin{array}{r} 0010001001 \\ \underset{10}{\oplus}\ 1110100100 \\ \hline 10000101101 \\ 1 \\ \hline 0000101110 \end{array}$$

REMARQUES. 1. Si $k > 1$, le dépassement de capacité de l'additionneur est signalé par l'apparition d'une combinaison inadmissible de zéros et d'unités dans les positions du signe du résultat. Pour $k = 1$, il y a dépassement lorsque le chiffre de poids le plus élevé du résultat diffère des poids les plus élevés des codes $[x]_{\text{inv}}$ et $[y]_{\text{inv}}$.

2. La soustraction des nombres se réduit aisément à l'addition moyennant la formule

$$x - y = x + (-y)$$

qui pour $|x - y| < 10^n$, implique la relation

$$[x - y]^k_{\text{inv}} = [x]^k_{\text{inv}} \underset{n+\nu+k}{\oplus} [-y]^k_{\text{inv}}.$$

2.2.5. Code complémentaire. Soit x un nombre à n chiffres dans la partie entière et ν chiffres dans la partie fractionnaire:

$$x = sa_{n-1} \ldots a_0, a_{-1} \ldots a_{-\nu}.$$

On appelle *code complémentaire* ou *complément vrai* $[x]^k_{\text{com}}$ du nombre x le nombre entier à $n + \nu + k$ chiffres défini par la formule

$$[x]^k_{\text{com}} = \begin{cases} \underbrace{0 \ldots 0}_{k \text{ zéros}} a_{n-1} \ldots a_0 a_{-1} \ldots a_{-\nu} \text{ pour } x \geqslant 0, \\ \underbrace{1 \ldots 1}_{k \text{ unités}} \bar{a}_{n-1} \ldots \bar{a}_0 \bar{a}_{-1} \ldots \bar{a}_{-\nu} \underset{n+\nu+k}{\boxplus} 0 \ldots 01 \text{ pour } x \leqslant 0, \end{cases} \quad (2.11)$$

où

$$\bar{a}_i = 1 - a_i \ (i = n-1, n-2, \ldots, 0, -1, \ldots, -\nu).$$

Les k poids forts du code complémentaire représentent le *signe*, les autres les *chiffres*. Pour k donné, le complément de zéro a la forme unique $[0]^k_{\text{com}} = 00 \ldots 0$.

Dans tout système de codage, la valeur de k est fixe.

Si l'on connaît le code complémentaire d'un nombre x, alors pour trouver le complément (vrai) du nombre $-x$ il faut remplacer le chiffre du code donné par son complément à l'unité et effectuer l'addition modulo $n + \nu + k$ du résultat obtenu avec le nombre $0 \ldots 01$.

En rapprochant les formules (2.8), (2.9) et (2.11) on voit que, pour des valeurs fixes de n, ν et k, les codes droits, inverse et complémentaire d'un nombre *positif* se confondent. Pour un nombre *négatif*, tous les trois codes sont différents.

Si les nombres binaires x et y satisfont à la condition (2.10), les formules (2.4) et (2.6) sont valables pour leurs codes complémentaires pour $m = n + \nu + k$; par conséquent, un additionneur sans report cyclique permet d'additionner les nombres représentés en code complémentaire. Pour $|x + y| \geqslant 10^n$ (10 est la notation binaire du nombre deux), la condition (2.6) n'est pas satisfaite et le résultat déborde la capacité de représentation *).

EXEMPLE 2.16. Supposons que $n = 0$, $\nu = 8$, $k = 2$ et soit $x = -0{,}10111010$, $y = 0{,}00111101$. Alors on a $x + y = -0{,}01111101$, donc $|x + y| < 10^n$. D'après la formule (2.11) on obtient $[x]^2_{\text{com}} = 1101000110$; $[y]^2_{\text{com}} = 0000111101$; $[x + y]^2_{\text{com}} = 1110000011$. En effectuant l'addition sans report cyclique des codes $[x]^2_{\text{com}}$ et $[y]^2_{\text{com}}$, on voit que le résultat est $[x + y]^2_{\text{com}}$:

$$\begin{array}{l r} \boxed{+} & 1101000110 \\ 10 & \underline{0000111101} \\ & 1110000011 \end{array}$$

EXEMPLE 2.17. Soit $n = 0$, $\nu = 10$, $k = 2$ et $x = -0{,}1010000001$, $y = -0{,}1100001001$. Dans ce cas on a $x + y = -1{,}0110001010$, $|x + y| > 10^n = 1$, donc $x + y$ n'a pas de code complémentaire et l'addition sans report cyclique des codes $[x]^2_{\text{com}} = 110101111111$ et $[y]^2_{\text{com}} = 110011110111$ conduit à un dépassement de capacité de l'additionneur.

En effet,

$$\begin{array}{l r l} & 110101111111 & \\ \boxed{+} & \underline{110011110111} & \\ 12 & 1101001110110 & (\text{«perte» de la retenue}) \\ & 101001110110 & \end{array}$$

Le résultat de l'addition sans report cyclique a une combinaison interdite 10 dans les positions du signe et n'est donc pas un code complémentaire.

REMARQUES. 1. Si le nombre de positions du signe d'un code complémentaire est $k > 1$, alors le dépassement de capacité se manifeste par l'apparition d'une combinaison interdite de zéros et d'unités dans les positions du signe du résultat. Pour $k = 1$, le dépassement de capacité se traduit par ce que les chiffres de rang

*) Plus exactement, le dépassement a lieu lorsque $x + y \geqslant 10^n$, ou bien lorsque $x + y < -10^n$.

le plus élevé des codes complémentaires des nombres initiaux et du résultat sont différents.

2. La soustraction des nombres se réduit aisément à l'addition, puisque

$$x - y = x + (-y)$$

implique

$$[x-y]^k_{\text{com}} = [x]^k_{\text{com}} \underset{n+\nu+k}{\boxed{+}} [-y]^k_{\text{com}}$$

pour $|x - y| < 10^n$.

2.2.6. Représentation des nombres en virgule fixe. Dans les calculateurs, les nombres en virgule fixe sont représentés par l'un des codes décrits plus haut : droit, inverse ou complémentaire. Dans certaines machines les différents organes traitent des nombres représentés en codes différents. Dans ce cas, le transfert des nombres entre organes suppose une conversion de code.

EXEMPLE 2.18. Considérons une machine traitant des nombres binaires en virgule fixe. Supposons qu'on réserve six positions pour la partie entière des nombres et sept positions pour la partie fractionnaire. Les nombres sont gardés en mémoire en code droit avec une position du signe. Quelques exemples de représentation des nombres dans la mémoire d'une telle machine sont donnés à la table 2.1. (Ici et plus bas la disposition du signe à gauche des chiffres du nombre n'est pas, en général, obligatoire.)

Table 2.1

Représentation des nombres en virgule fixe en code droit

Notation décimale	Notation binaire	Représentation du nombre dans la machine			Code machine
		signe	partie entière	partie fractionnaire	
+21,75	+10101,11	0	010101	1100000	00101011100000
−4,125	−100,001	1	000100	0010000	10001000010000
−0,203125	−0,001101	1	000000	0011010	10000000011010

EXEMPLE 2.19. Une machine à virgule fixe traite des nombres binaires. La virgule est située juste à droite du signe ce qui correspond à des nombres fractionnaires. La partie fractionnaire est de dix positions. Dans l'unité de calcul les nombres sont représentés

par leurs codes inverses avec deux positions du signe. Quelques exemples de représentation sont donnés à la table 2.2.

Table 2.2

Représentation des nombres en virgule fixe en code inverse

Notation décimale	Notation binaire	Représentation du nombre dans la machine		Code machine
		signe	partie fractionnaire	
+0,203125	+0,001101	00	0011010000	000011010000
−0,203125	−0,001101	11	1100101111	111100101111
+0	+0	00	0000000000	000000000000
−0	−0	11	1111111111	111111111111

EXEMPLE 2.20. Une machine à virgule fixe traite des nombres binaires représentés en code complémentaire avec deux positions du signe. La virgule est fixée juste à droite du signe, la partie fractionnaire a neuf positions.

Quelques exemples de représentation des nombres dans l'unité de calcul d'une telle machine sont donnés à la table 2.3.

Table 2.3

Représentation des nombres en virgule fixe en code complémentaire

Notation décimale	Notation binaire	Représentation du nombre dans la machine		Code machine
		signe	partie fractionnaire	
+0,6875	+0,1011	00	101100000	00101100000
−0,6875	−0,1011	11	010100000	11010100000

2.2.7. Langage des nombres en virgule flottante. Représentation des nombres dans le calculateur. Un langage des nombres en virgule flottante correspondant aux nombres normalisés en virgule flottante s'obtient en complétant les formules de Backus du p. 2.2.1. par les formules suivantes:

⟨mantisse⟩ :: = ⟨signe⟩ ⟨partie fractionnaire⟩

⟨exposant⟩ :: = ⟨signe⟩ ⟨partie entière⟩

⟨nombre en virgule flottante⟩ :: = ⟨mantisse⟩ ⟨exposant⟩ .

Le format d'un nombre en virgule flottante a la forme

$$\underbrace{\langle\text{signe}\rangle}_{k_1}\underbrace{\langle\text{partie fractionnaire}\rangle}_{\nu}\underbrace{\langle\text{signe}\rangle}_{k_2}\underbrace{\langle\text{partie entière}\rangle}_{n}$$

Lorsqu'il s'agit des nombres en virgule flottante, on dit « valeur absolue de la mantisse » et « valeur absolue de l'exposant » au lieu de « partie fractionnaire » et « partie entière » respectivement.

Dans le calculateur, les nombres en virgule flottante sont représentés sous forme d'une réunion de deux codes: celui de la mantisse et celui de l'exposant. On admet une même convention ou des conventions différentes de représentation de la mantisse et de l'exposant. Par exemple, on peut choisir le code droit pour la mantisse et le code complémentaire pour l'exposant, etc. Les différents organes du calculateur peuvent être adaptés à des codes différents. Ainsi, dans la mémoire rapide de certaines machines, les mantisses et les exposants des nombres sont représentés en code droit avec une position du signe, tandis que dans l'unité de calcul ils sont représentés en convention du complément restreint avec deux positions du signe.

EXEMPLE 2.21. Soit un calculateur à virgule flottante, traitant des nombres en code droit avec une position du signe pour les exposants et pour les mantisses. Sept positions binaires sont réservées pour représenter la mantisse et quatre pour représenter l'exposant, sans compter celles du signe. Quelques exemples de représentation des nombres (normalisés) dans la mémoire rapide d'une telle machine sont donnés à la table 2.4. (La disposition du signe à gauche des

Table 2.4

Représentation des nombres en virgule flottante en code droit

Notation décimale	Forme normalisée du nombre binaire	Représentation du nombre dans la machine				Code machine
		signe de la mantisse	mantisse	signe de l'exposant	exposant	
+11,25	$+0{,}101101\cdot 10^{+100}$	0	1011010	0	0100	0101101000100
−11,25	$-0{,}101101\cdot 10^{+100}$	1	1011010	0	0100	1101101000100
+0,23046875	$+0{,}111011\cdot 10^{-10}$	0	1110110	1	0010	0111011010010
−0,23046875	$-0{,}111011\cdot 10^{-10}$	1	1110110	1	0010	1111011010010
−0,1	$-0{,}(1100)\cdot 10^{-11}$ *)	1	1100110	1	0011	1110011010011

*) La notation 0,(1100) désigne une fraction binaire périodique de période 1100.

chiffres de la mantisse ou de l'exposant n'est en général pas obligatoire.)

EXEMPLE 2.22. Considérons un calculateur traitant des nombres en virgule flottante, dont les mantisses sont représentées en code droit avec une position du signe, et les exposants sont représentés en code complémentaire avec également une position du signe. La virgule est placée juste après le signe. Il est prévu huit positions pour la mantisse et quatre pour l'exposant, sans compter celles du signe. Quelques exemples de représentation des nombres normalisés dans la mémoire rapide d'une telle machine sont donnés à la table 2.5.

Table 2.5

Représentation des nombres en virgule flottante (les mantisses sont représentées en code droit, les exposants, en code complémentaire)

Notation décimale	Forme normalisée du nombre binaire	Représentation du nombre dans la machine				Code machine
		signe de la mantisse	mantisse	signe de l'exposant	exposant	
+12,75	$+0{,}110011 \cdot 10^{+100}$	0	11001100	0	0100	01100110000100
−0,140625	$-0{,}1001 \cdot 10^{-10}$	1	10010000	1	1110	11001000011110

2.2.8. Codage de l'information alphanumérique.

Dans certains calculateurs, les entiers décimaux peuvent être représentés sous forme d'une suite arbitrairement longue formée des codes de chiffres. Habituellement, le code de chaque chiffre décimal est un nombre binaire de quatre chiffres qui a la même valeur. Les combinaisons binaires non utilisées servent à la représentation du signe (voir la table 2.6).

Table 2.6

Représentation binaire des chiffres décimaux

Chiffre décimal	Code	Chiffre décimal	Code	Signe	Code
0	0000	6	0110	+	1010 } ASCII
1	0001	7	0111	−	1011 } ASCII
2	0010	8	1000	+	1100 } IBB
3	0011	9	1001	−	1101 } IBB
4	0100			+	1110
5	0101			−	1111

Dans les différents systèmes de codage, on choisit de manière différente les codes des signes. Ainsi, dans le code EBCDIC (Extended Binary Coded Decimal Interchange Code) employé par la firme IBM (U.S.A), le signe + est représenté par 1100 et le signe —, par 1101. Dans le code ASCII (American Standard Code for Information Interchange) le signe + est traduit par 1010 et le signe —, par 1011.

Les caractères de l'information alphanumérique sont codés chacun par un nombre binaire de huit chiffres. Avec cette méthode on peut coder jusqu'à 256 caractères.

Le format d'un nombre binaire d'un seul chiffre est souvent appelé bit, et le format d'un nombre binaire de huit chiffres s'appelle octet. Un octet vaut huit bits.

La table 2.7 donne un fragment du code alphanumérique EBCDIC.

Table 2.7

Fragment du code alphanumérique EBCDIC à huit moments (la première colonne contient les poids forts d'un octet, et la première ligne, les poids faibles)

	1000	1001	1000	1011	1100	1101	1110	1111
0000					>	<	≠	0
0001	*a*	*j*			*A*	*J*		1
0010	*b*	*k*	*s*		*B*	*K*	*S*	2
0011	*c*	*l*	*t*		*C*	*L*	*T*	3
0100	*d*	*m*	*u*		*D*	*M*	*U*	4
0101	*e*	*n*	*v*		*E*	*N*	*V*	5
0110	*f*	*o*	*w*		*F*	*O*	*W*	6
0111	*g*	*p*	*x*		*G*	*P*	*X*	7
1000	*h*	*q*	*y*		*H*	*Q*	*Y*	8
1001	*i*	*r*	*z*		*I*	*R*	*Z*	9

Les codes des informations alphanumériques peuvent être arbitrairement longs (suivant le texte codé).

§ 2.3. Langages algorithmiques de machine

2.3.1. Opérations élémentaires de machine. En principe, tout calculateur est conçu en vue de résoudre les problèmes d'une certaine classe. Il faut donc que le calculateur sache effectuer, en combinaisons demandées, certaines opérations élémentaires. Pour toute opération élémentaire, on construit une opération élémentaire de machine

qui en est assez proche et qui a un caractère local au sens du p. 1.5.6. Voyons ces dernières opérations de plus près.

Les opérations élémentaires de machine sont : l'entrée de l'information dans la mémoire principale à partir d'une mémoire quelconque, la sortie de l'information de la mémoire principale, ainsi que toute opération qui :

— est réalisée par les circuits du calculateur (câblée) ;

— a pour données initiales les résultats d'opérations élémentaires de machine enregistrés dans une ou plusieurs cases de mémoire ;

— fournit un résultat qui est enregistré dans une case de mémoire spécialement réservée et qui est disponible (mais pas forcément utilisé) en tant que donnée initiale pour une opération élémentaire de machine ;

— ne peut être considérée comme une combinaison de plusieurs opérations de machine plus simples qui satisfont à trois conditions précédentes.

L'ensemble de toutes les opérations de machine prévues pour un calculateur commandé par programmes s'appelle *système d'opérations* de ce calculateur. Il faut faire une nette distinction entre la notion d'instruction et celle d'opération. Une *instruction* est une prescription élémentaire qui prévoit l'exécution d'un groupe d'opérations.

En classant les opérations de machine il faut tenir compte du fait que leur nature et le rôle qu'elles jouent dépendent essentiellement de l'endroit où se trouvent leurs données initiales, et surtout de l'endroit où leurs résultats sont inscrits. Cela est dû à l'existence de plusieurs dispositifs de mémoire qui jouent dans le fonctionnement du calculateur des rôles principalement différents.

2.3.1.1. Classification des mémoires du calculateur. La mémoire principale, les mémoires externes, les registres de l'unité de calcul, les cartes et rubans perforés sont destinés à garder les codes qui représentent des nombres et des instructions (désignons tous ces dispositifs par la lettre A). Le *registre d'instruction* (désignons-le par B) sert à recevoir l'instruction à exécuter et qui détermine donc les prochaines actions du calculateur. Lors de l'exécution de chaque instruction, dans un moment déterminé, le *compteur ordinal* (désignons-le par C) doit contenir l'adresse de l'instruction suivante. (Dans les calculateurs où l'ordre d'exécution des instructions est libre, le compteur ordinal fait généralement partie du registre d'instruction.) Les mémoires à une position (désignons leur ensemble par D) sont destinées à garder les signaux spéciaux (par exemple, le signal qu'on appelle « signal ω ») utilisés pour le contrôle automatique de l'ordre d'exécution des instructions. Dans certains calculateurs il y a des *registres d'index* et *de base* (désignons-les par F) dont le contenu peut s'additionner à celui du registre d'instruction (ou

s'en soustraire) ce qui a pour effet de modifier l'instruction en cours d'exécution. Habituellement, dans les machines pourvues de registres d'index, une mémoire à une position est prévue pour recevoir un signal spécial qui se produit lorsque le contenu du registre d'index satisfait à une certaine condition, par exemple s'annule. Ces mémoires à une position font partie de D.

Un calculateur plus évolué peut posséder d'autres dispositifs spéciaux que nous omettrons de mentionner ici. Les dispositifs A, B, C, D et F sont présents dans tous les calculateurs actuels. Les dispositifs B, C, D et F diffèrent selon le type du calculateur et sont appelés différemment.

2.3.1.2. Classification des opérations de machine. Rangeons dans la première classe toutes les opérations dont les opérandes sont contenus dans A et dont les résultats sont enregistrés dans A (classe d'opérations « $A \rightarrow A$ »). Cette classe comporte les opérations *arithmétiques*, les opérations dites *non arithmétiques*, les *transferts* et certaines autres opérations sur des nombres, des codes et des instructions.

La deuxième classe (classe « $\rightarrow D$ ») renferme les opérations dont les résultats sont enregistrés dans D. Ces opérations sont dites *logiques* puisqu'elles vérifient si les nombres, les codes ou les instructions qui en font l'objet vérifient certaines conditions.

Nous rapportons à la troisième classe (classe « $\rightarrow C$ ») les opérations dont les résultats s'enregistrent dans C. Il s'agit d'opérations d'organisation du programme (appelées *branchements* ou *sauts*). Elles ont pour but de modifier le déroulement séquentiel des instructions.

Dans la quatrième classe (classe « $\rightarrow B$ ») nous réunissons les opérations dont les résultats sont enregistrés dans B. On y trouve les opérations qui permettent de lire une instruction dans la mémoire principale et de modifier le contenu du registre d'instruction.

Toutes les autres opérations de machine, sauf celle d'arrêt, seront rapportées à la cinquième classe. En particulier, on y classera les opérations dont les résultats sont enregistrés dans F.

L'opération d'arrêt qui consiste en la transformation identique du contenu de toutes les mémoires du calculateur sera classée à part. En pratique cette opération signifie l'arrêt du fonctionnement du calculateur.

Les opérations que nous venons d'évoquer n'épuisent certes pas tous les types d'opérations réalisées dans les calculateurs actuels.

Signalons que, dans certains calculateurs modernes, l'opération d'arrêt n'est pas prévue. Cela ne veut point dire que, pour arrêter le processus de traitement de l'information on est obligé de couper brutalement l'alimentation électrique! Tout simplement, l'opération d'arrêt est réalisée par programme (v. p. 2.3.9).

2.3.2. Opérations arithmétiques. Les algorithmes de réalisation des opérations arithmétiques sont différents dans les calculateurs différents et ils déterminent la conception des organes de calcul de ces derniers. La méthode d'exécution de ces opérations dépend de la forme de représentation des nombres dans la machine.

Addition et soustraction des nombres en virgule fixe. Les opérandes des opérations d'addition et de soustraction en virgule fixe se représentent habituellement en code inverse ou en code complémentaire (selon le type de l'additionneur), et les opérations se réalisent d'après les règles du § 2.2. Lorsque la mémoire garde ces nombres dans un autre code, une conversion dans le code nécessaire s'impose. S'il y a un débordement de capacité, un signal spécial est envoyé par l'additionneur à l'unité de commande. Ce signal que l'on appelle d'ordinaire signal φ représente un nombre binaire à un chiffre. Ainsi, $\varphi = 1$ si le débordement de capacité a lieu, et $\varphi = 0$ dans le cas contraire.

Addition et soustraction des nombres en virgule flottante. L'addition ou la soustraction en virgule flottante nécessite la dénormalisation du plus petit des deux opérandes pour l'amener à avoir même exposant que l'autre, puis éventuellement la normalisation du résultat!

Supposons que les deux opérandes soient normalisés et donnés sous la forme de couples de nombres m_1, n_1 et m_2, n_2. Pour fixer les idées, supposons que $n_2 > n_1$. Alors l'opérande représenté par le couple m_1, n_1 se réduit à la forme $m_1 \cdot 10^{n_1 - n_2}$, n_2 ou, comme on dit, est dénormalisé avec le facteur de dénormalisation

$$r = n_1 - n_2.$$

Après dénormalisation du nombre de plus petit exposant l'addition des mantisses se fait comme l'addition des nombres en virgule fixe. Trois cas peuvent se présenter lors de l'addition des mantisses.

1. La somme $m = m_1 \cdot 10^{n_1 - n_2} + m_2$ satisfait à la condition $\frac{1}{10} \leqslant |m| < 1$. Alors le résultat de l'addition se conserve sous la forme du couple de nombre m, n_2.

2. Le nombre m satisfait à la condition $|m| < \frac{1}{10}$ (i.e. une dénormalisation « à droite » s'est produite lors de l'addition). Dans ce cas, le résultat m, n_2 subit une normalisation avant d'être enregistré en tant que somme.

3. L'addition des nombres $m_1 \cdot 10^r$ et m_2 conduit à un dépassement de capacité des positions de l'additionneur réservées aux mantisses (on dit alors qu'il y a une dénormalisation « à gauche »). Dans ce cas on effectue une correction du résultat qui consiste à décaler tous les chiffres du code obtenu dans l'additionneur d'une position vers la droite. Le décalage à droite permet de conserver le poids le plus fort du code, le poids le plus faible du code retenu est arrondi

de manière habituelle. En même temps on fait augmenter l'exposant d'une unité. Une telle correction du résultat est équivalente à ceci: on obtient pour somme le nombre normalisé $(m_1 \cdot 10^r + m_2) \cdot 10^{-1}$, $n_2 + 1$.

Le dépassement de capacité de la partie de l'additionneur réservée aux mantisses n'altère pas le résultat. Le dépassement de capacité de tout l'additionneur, au cours d'une addition ou d'une soustraction, ne se produit que lorsque l'exposant n_2 augmente d'une unité et consiste en le dépassement de la partie de l'additionneur réservée à l'exposant. Dans ce cas l'additionneur fournit à l'unité de commande le signal de dépassement qui est appelé, comme pour les machines à virgule fixe, signal φ ($\varphi = 1$ s'il y a dépassement, $\varphi = 0$ dans le cas contraire).

Multiplication et division des nombres. La multiplication des nombres en virgule flottante ou en virgule fixe placée devant le poids le plus fort s'effectue généralement en code droit.

Tout d'abord, on additionne les signes des nombres dans un additionneur à un chiffre. La retenue, si elle se forme est perdue. Le signe attribué au produit se détermine d'après la règle algébrique usuelle.

La multiplication proprement dite s'effectue de la même manière sur des nombres représentés en virgule fixe et sur les mantisses des nombres en virgule flottante.

Dans le système binaire, le multiplicande n'est multiplié que par les zéros ou les unités qui sont des chiffres du multiplicateur. Les produits partiels ainsi obtenus sont écrits l'un au-dessous de l'autre, avec déplacement de chaque nouveau produit d'une position. On omet d'écrire les produits nuls. Ceci vu, une multiplication dans un calculateur peut être réalisée par une succession d'additions des produits partiels et de décalages (le nombre de ces derniers est déterminé par les chiffres du multiplicateur). Dans certains calculateurs, la multiplication commence par les poids forts du multiplicateur.

Exemple 2.23. Un exemple de multiplication qui commence par les poids forts du multiplicateur:

×	0,11001101		
	0,11101001		
	01100110	1	résultat du premier décalage
	00110011	01	résultat du deuxième décalage, etc.
+	00011001	101	
	00000110	01101	
	00000000	11001101	
	0,10111010	10010101	
Arrondissement:	0,10111011		

Dans une machine à virgule flottante, on fait l'addition algébrique des exposants des deux opérandes ce qui fournit l'exposant du produit. Si le résultat de la multiplication en virgule flottante est non normalisé, il doit être normalisé (avant l'arrondissement).

Dans le cas d'une division, le signe du quotient s'obtient en sommant dans un additionneur à un chiffre, de même que pour la multiplication les chiffres qui représentent les signes du dividende et du diviseur.

La division des nombres en virgule fixe et la division des mantisses dans les machines à virgule flottante se font pratiquement de la même façon. On soustrait le diviseur du dividende, on décale le reste obtenu à gauche, puis on soustrait une nouvelle fois le diviseur du reste décalé, etc.

Dans les machines à virgule flottante, on effectue une soustraction algébrique de l'exposant du diviseur de celui du dividende; si le quotient s'avère être non normalisé, on opère sa normalisation.

Les opérations sur les exposants des opérandes peuvent donner lieu à des débordements de capacité. Alors l'organe de calcul envoie à l'organe de commande le signal $\varphi = 1$.

2.3.3. Opérations modulo deux, transferts. Convenons de numéroter les chiffres d'un mot machine de droite à gauche par les nombres $0, 1, 2, \ldots, n-1, n$ (notons qu'il existe différents modes de numérotation des positions d'une cellule). Le chiffre binaire occupant la i-ème place dans un code a, b ou c sera désigné respectivement par a_i, b_i ou c_i.

Décalage. Nous appelons décalage de k positions (k entier) d'un code

$$a = a_n a_{n-1} \ldots a_1 a_0$$

l'opération définie par la formule

$$c_i = \begin{cases} a_{i-k} & \text{pour } 0 \leqslant i-k \leqslant n, \\ 0 & \text{pour les autres valeurs de } i-k \end{cases}$$

où c_i ($i = 0, 1, \ldots, n$) sont les chiffres du résultat.

Parfois, pour $k > 0$, on appelle cette opération décalage de k positions *à gauche*, et pour $k < 0$, décalage de $|k|$ positions *à droite.*

EXEMPLE 2.24. Un décalage de trois positions (pour $n = 7$) du nombre

$$a = 10011101$$

donne

$$c = 11101000.$$

Le décalage du même code a de —3 positions donne

$$c' = 00010011.$$

Multiplication logique. La multiplication logique de deux codes binaires

$$a = a_n a_{n-1} \ldots a_1 a_0,$$
$$b = b_n b_{n-1} \ldots b_1 b_0$$

s'effectue d'après la formule

$$c_i = a_i \wedge b_i \quad (i = 0, 1, \ldots, n).$$

On peut utiliser cette opération pour « dégager » les différents « morceaux » de l'un des codes en choisissant convenablement l'autre.

EXEMPLE 2.25. Soit $n = 7$ et

$$a = 10111001.$$

Si a est le code droit d'un nombre x (négatif dans notre cas), alors, pour dégager le signe du nombre x, il faut choisir comme b le code $b = 10000000$. On aura $c = 10000000$. Pour dégager la valeur absolue du nombre x, il faut poser $b' = 01111111$. On aura $c' = $ $= 00111001$.

Addition logique. L'addition logique de deux codes a et b est définie par la formule

$$c_i = a_i \vee b_i \quad (i = 0, 1, \ldots, n).$$

Cette opération permet de « former » un code à partir de deux codes gardés dans les différentes cases de mémoire.

EXEMPLE 2.26. Soit $n = 8$. Supposons que $a = 110100000$, $b = $ $= 000001010$. L'addition logique des codes a et b donne $c = $ $= 110101010$.

Comparaison. On appelle comparaison la négation logique de l'équivalence de deux codes a et b définie par formule

$$c_i = a_i \not\sim b_i \quad (i = 0, 1, \ldots, n).$$

Si les positions i des codes a et b contiennent le même chiffre, alors la position i du résultat sera zéro. Si les chiffres des positions i des codes a et b sont différents, la position i du résultat contiendra l'unité.

EXEMPLE 2.27 Soit $n = 7$. Supposons que $a = 11010001$, $b = $ $= 10011101$. En comparant ces codes nous aurons $c = 01001100$.

Ce résultat montre que les positions 0, 1, 4, 5 et 7 des codes comparés ont des chiffres égaux, tandis que les positions 2, 3 et 6 ont les chiffres différents.

Transferts. Les opérations de transfert sont définies par la formule

$$c_i = a_i \quad (i = 0,\ 1,\ \ldots,\ n), \tag{2.12}$$

la case contenant le code initial a, de même que la case recevant ce code (on le désigne alors c) appartiennent aux dispositifs A. Les transferts de codes peuvent être effectués:

— entre les cases de mémoire principale,

— entre un groupe de cases de mémoire principale et une zone de bande, de tambour ou de disque magnétiques;

— entre la mémoire principale et les supports d'information (bandes de papier, cartes et rubans perforés): c'est la sortie des résultats;

— entre les supports d'information (cartes ou rubans perforés) et la mémoire principale: c'est l'entrée de données.

Dans les machines où l'additionneur est utilisé non seulement pour réaliser des opérations, mais aussi pour garder les opérandes, on prévoit des transferts entre la mémoire principale et l'additionneur.

2.3.4. Opérations logiques. Nous appelons opérations logiques les opérations de calcul de prédicats et dont le résultat est enregistré dans D (i.e. dans l'un des registres à une position destinés à garder des signaux). On peut répartir les opérations logiques en groupes selon l'origine de leurs données initiales. L'un de ces groupes est composé des opérations de la forme $A \rightarrow D$ dont les données initiales se trouvent dans l'additionneur ou dans un autre registre de l'organe de calcul. Les résultats de ces opérations sont habituellement appelés valeurs du signal ω. L'une ou plusieurs opérations logiques de ce groupe s'appellent formation du signal φ (signal de débordement de capacité). Un autre groupe d'opérations logiques est composé des opérations de la forme $F \rightarrow D$ dont les données initiales proviennent des registres d'index. Les opérations logiques utilisées dans les calculateurs sont très variées. Voyons certaines d'entre elles; les codes seront représentés en convention du paragraphe précédent.

Test du signe. On appelle test du signe l'opération logique de la forme $A \rightarrow D$ définie par la formule

$$\omega = a_n.$$

Elle calcule le prédicat qui dit: « le chiffre de la n-ème position du code a est 1 ». Le nom de l'opération est dû au fait que, dans le cas où a est le code droit ou inverse d'un nombre x, elle conduit à la formation du signal $\omega = 1$ pour $x \leqslant -0$ et du signal $\omega = 0$ pour

$x \geqslant +0$. Si a est le code complémentaire du nombre x, alors cette opération donne $\omega = 0$ pour $x \geqslant 0$ et $\omega = 1$ pour $x < 0$.

Test de débordement de capacité. Le test de débordement de capacité lors d'une addition appartient au groupe d'opérations $A \to D$ et, dans le cas où l'on utilise le code inverse ou complémentaire avec deux positions du signe, a la forme

$$\varphi = a_n \not\sim a_{n-1}.$$

Dans certains calculateurs le signal φ est utilisé pour modifier l'ordre d'exécution des instructions. Dans d'autres machines le signal $\varphi = 1$ arrête le fonctionnement.

Test du zéro. Cette opération représente le calcul du prédicat

$$\omega = \overline{a_n \vee \ldots \vee a_1 \vee a_0}.$$

Le résultat vaut 1 lorsque $a_i = 0$ $(i = 0, 1, \ldots, n)$ et 0 lorsque le code a contient le chiffre 1 dans au moins une position.

Si cette opération appartient au groupe $A \to D$, on peut s'en servir pour savoir quel est le résultat d'une comparaison. Elle donnera $\omega = 1$ si les codes comparés sont identiques et $\omega = 0$ s'ils diffèrent par au moins un chiffre. On peut utiliser cette opération logique pour vérifier si le code contenu dans un registre d'index est identiquement nul (dans ce cas elle appartiendra au groupe $F \to D$).

Test de valeur absolue. Dans les calculateurs à virgule flottante on utilise souvent cette opération logique. Désignons par μ le numéro de la position réservée au signe de l'exposant d'un nombre normalisé. Alors l'opération en question peut être donnée comme prédicat

$$\omega = \bar{a}_\mu \wedge (a_{\mu-1} \vee a_{\mu-2} \vee \ldots \vee a_0).$$

Il est aisé de comprendre que cette opération fournit le signal $\omega = 1$ si, et seulement si, les deux conditions suivantes sont satisfaites:

1) la position du signe de l'exposant contient le chiffre 0;

2) l'une au moins des positions réservées à la valeur absolue de l'exposant contient l'unité.

Si a est le code d'un nombre normalisé x, l'exposant de ce nombre vérifie l'inégalité $p \geqslant 1$ si $\omega = 1$ (et seulement sous cette condition). Il en découle que le signal $\omega = 1$ traduit le fait que

$$|x| = |m \cdot 10^p| \geqslant \frac{1}{10} \cdot 10^1 = 1.$$

La formule obtenue

$$|x| \geqslant 1$$

explique la signification et le nom de l'opération.

2.3.5. Opérations modifiant le contenu du registre d'instruction. Les opérations qui modifient le contenu du registre d'instruction se rapportent à la classe $\to B$ et sont très variées. Tout comme les

opérations logiques, elles peuvent être réparties en plusieurs groupes selon l'origine de leurs données initiales.

Extraction de l'instruction. Cette opération se rapporte à la classe $A \to B$ et représente un transfert du code en conformité de la formule (2.12). Elle diffère des transferts déjà décrits par le fait que son résultat est envoyé non pas dans l'un des dispositifs A, mais dans le registre d'instruction (B). Le contenu de ce registre détermine le fonctionnement du calculateur pendant un certain temps. L'instruction est lue dans la case de mémoire principale dont le numéro est contenu dans le compteur ordinal. Disons que c'est une opération d'extraction de l'instruction de Iʳᵉ espèce.

Dans certains calculateurs, le registre d'instruction représente un additionneur. Dans ces machines les opérations d'extraction d'instructions appartiennent à la classe d'opérations A, $B \to B$ et représentent une addition du code se trouvant dans une case de mémoire (A) avec l'ancien contenu du registre d'instruction (B) avec l'enregistrement du résultat dans le registre d'instruction ($\to B$). Dans la plupart des cas, l'ancien contenu du registre d'instruction est nul. Cette opération sera appelée extraction de l'instruction de IIᵉ espèce.

Modification d'instructions. L'opération de modification d'instructions est utilisée dans les machines où l'on adopte l'extraction de l'instruction de IIᵉ espèce, elle s'effectue avant cette dernière et se rapporte à la classe $A \to B$. Elle représente le transfert du code d'une case de mémoire principale dans le registre d'instruction. On prend des mesures appropriées pour préserver le contenu du registre d'instruction jusqu'à l'appel de l'instruction suivante.

Modification de l'instruction par le contenu du registre d'index. Cette opération est prévue dans les calculateurs munis de registres d'index. Elle est du type B, $F \to B$ et consiste en ce que le contenu d'un registre d'index s'ajoute ou se soustrait de celui du registre d'instruction, le résultat étant inscrit dans le registre d'instruction. Les adresses que l'on obtient de la sorte sont dites *effectives*.

2.3.6. Branchements. Les opérations d'organisation du programme auxquelles se rapportent les branchements ont cette particularité que leurs résultats s'enregistrent dans le compteur ordinal C. Après avoir considéré les opérations d'extraction d'instructions, on comprendra facilement la signification des opérations de branchement, compte tenu du fait que le numéro de l'instruction à extraire de la mémoire est déterminé par le contenu du compteur ordinal.

Branchements inconditionnels. Dans les calculateurs où l'ordre d'exécution d'instructions est donné, il existe deux types de branchements inconditionnels. Le premier représente la progression d'une unité du contenu du compteur ordinal (opération de la forme $C \to C$).

Cette opération assure une exécution séquentielle des instructions du programme.

Le deuxième type de branchements inconditionnels représente le transfert de l'adresse de l'instruction de B (registre d'instruction) dans C (compteur ordinal).

Dans les machines où l'ordre d'exécution des instructions est libre, cette opération se fait en simultanéité avec l'extraction de l'instruction, puisque le compteur ordinal dans ces machines fait partie du registre d'instruction.

Branchements conditionnels. Dans les machines actuelles avec l'ordre d'exécution des instructions donné, on trouve des branchements conditionnels à une et à deux adresses.

Un branchement conditionnel à une adresse représente l'opération de la forme

$$c=\begin{cases} b & \text{pour } \omega=1, \\ a+1 & \text{pour } \omega=0, \end{cases} \tag{2.13}$$

où a est le contenu du compteur ordinal avant l'exécution de cette opération, b, la partie adresse de l'instruction se trouvant dans le registre d'instruction, ω le contenu du registre D. Le résultat de cette opération est placé dans le compteur ordinal. Ainsi, pour $\omega = 0$ le contenu du compteur ordinal augmente d'une unité, et pour $\omega = 1$ la partie adresse de l'instruction se trouvant dans le registre d'instruction est introduite dans le compteur ordinal.

Dans certains calculateurs, en plus d'un branchement conditionnel défini par (2.13) ou à sa place, on prévoit un branchement conditionnel à une adresse qui correspond à la formule

$$c=\begin{cases} a+1 & \text{pour } \omega=1, \\ b & \text{pour } \omega=0. \end{cases}$$

Pour les machines à une adresse sont possibles seuls les branchements conditionnels à une adresse. Il existe pourtant des machines à trois adresses utilisant cette opération.

Un branchement conditionnel à deux adresses représente l'opération définie par la formule

$$c=\begin{cases} a & \text{pour } \omega=1, \\ b & \text{pour } \omega=0, \end{cases}$$

où a et b sont les adresses figurant dans l'instruction gardée dans le registre d'instruction (B), et ω est le contenu de D. Le résultat de cette opération est inscrit dans le compteur ordinal C.

A la suite de cette opération, la commande passe à l'instruction gardée dans la case de numéro a si $\omega = 1$, et à l'instruction d'adresse b, si $\omega = 0$.

Remarque. En rapport avec les opérations de branchement, il faut mentionner l'opération qui forme l'instruction de retour. Il s'agit de former, à partir du contenu du compteur ordinal, une instruction de branchement inconditionnel et de la stocker dans la case appropriée de la mémoire principale. Cette opération est du type $C \to A$.

2.3.7. Opérations modifiant le contenu de registres d'index. Ces opérations se rapportent à la sixième classe. Nous n'en décrivons que quelques-unes.

Chargement du registre d'index. Dans certaines machines on prévoit le transfert d'une partie du code se trouvant dans le registre d'instruction dans un registre d'index (opération du type $B \to F$) ; dans d'autres machines on extrait l'information nécessaire d'une case de mémoire principale pour la placer dans le registre d'index (opération du type $A \to F$). Les deux opérations s'appellent chargement du registre d'index.

Modification du contenu du registre d'index. Cette opération est du type $F \to F$ et consiste à ajouter (ou à soustraire) au contenu d'un registre d'index une constante et à enregistrer le résultat dans le registre d'index.

2.3.8. Opérations sur l'information alphabétique. Pour la plupart des calculateurs, les opérations sur l'information alphabétique sont complexes et représentent des combinaisons d'opérations décrites plus haut effectuées sur des chaînes de codes (de mots machine) et sur leurs parties. La complexité provient du fait que les mots dont se compose l'information alphabétique dépassent le format du mot machine.

Les opérations principales sur l'information alphabétique sont:

— détermination de la longueur de mots;
— transfert d'un mot entre les cases de la mémoire principale;
— sélection d'une partie déterminée d'un mot donné;
— formation d'espaces blancs entre les mots;
— segmentation d'une suite de mots en parties plus petites;
— comparaison de deux mots.

Les opérations citées font partie du travail d'édition. Les opérations sur l'information alphabétique peuvent être élémentaires dans les machines où les cases de mémoire et les mots machine sont de longueur variable (l'adresse d'un mot machine est donnée par un couple de nombres: le numéro de l'octet de mémoire principale et la longueur du mot machine; dans les machines de ce genre tous les octets de mémoire principale sont numérotés successivement), ou dans les machines à cases de longueur fixe (d'ordinaire assez réduite) mais où les mots machine sont de longueur variable (l'adresse d'un mot machine est donnée par un couple de nombres: le numéro de la

case où est rangé le début du mot machine et la quantité de cases réservées à ce mot).

2.3.9. Langages algorithmiques de machine. La morphème principale d'un langage algorithmique de machine est le ⟨code d'opération⟩.

$$\langle\text{code d'opération}\rangle ::= \underbrace{\langle\text{lettre}\rangle\,\langle\text{lettre}\rangle\ \ldots\ \langle\text{lettre}\rangle}_{p\ \text{lettres}}$$

De plus, on considère habituellement les morphèmes ⟨adresse⟩, ou même plusieurs types de telles morphèmes : ⟨indice⟩, ⟨index⟩, etc.

A partir de ces morphèmes qui sont des mots dans un alphabet à deux lettres, en les disposant dans un certain ordre, on forme un mot appelé ⟨instruction⟩.

Généralement, la position et la longueur de la morphème ⟨code d'opération⟩ dans le format d'une ⟨instruction⟩ sont strictement déterminées. La disposition, la quantité et la longueur d'autres morphèmes dans une ⟨instruction⟩ dépendent du code d'opération.

⟨mot machine⟩ :: = ⟨instruction⟩ | ⟨opérande de machine⟩
⟨adresse propre⟩ :: = ⟨adresse⟩
⟨ordre⟩ :: = ⟨adresse propre⟩ | ⟨instruction⟩
⟨programme⟩ :: = ⟨ordre⟩ | ⟨programme⟩ ⟨ordre⟩

Le programme est la construction principale (notation d'un algorithme) dans le langage machine. Les ⟨instructions⟩ d'un programme introduit dans le calculateur sont disposées dans les cases de mémoire dont les numéros coïncident avec les ⟨adresses propres⟩ des instructions. Il en résulte que chaque adresse propre ne se rencontre qu'une seule fois dans un programme ayant un sens.

Une particularité des langages algorithmiques de machine est qu'à tout code d'opération correspond non pas une seule opération élémentaire de machine, mais tout un groupe d'opérations. Les opérandes d'opérations élémentaires sont pris dans les cases de mémoire dont les adresses occupent une place déterminée dans l'instruction ; d'autres positions de l'instruction sont réservées aux adresses des résultats. En même temps, outre l'opération nécessaire à la résolution du problème traité, quelques autres opérations se réalisent (on ne tient pas compte de leurs résultats lors de l'exécution ultérieure du programme). Les morphèmes ⟨indices⟩ et ⟨index⟩ précisent les ⟨codes d'opérations⟩ et ⟨adresses⟩.

En écrivant les ordres sur des formulaires, au lieu d'écrire les morphèmes dans un alphabet à deux lettres, par exemple dans $\{0, 1\}$, on les représente à l'aide de nombres d'un système de numération de base supérieure à deux. Le plus souvent on utilise les systèmes de bases huit ou seize, alors un chiffre octal désigne un groupe de

trois lettres binaires et un chiffre hexadécimal, un groupe de quatre lettres binaires. Ainsi, les chiffres octaux 4 et 7 désignent les groupes 100 et 111 respectivement, et les mêmes chiffres de base seize, les groupes 0100 et 0111. L'exécution d'un algorithme écrit dans un langage machine est généralement très simple.

L'essentiel de cet algorithme, en grands traits, consiste en ce que la valeur d'une quantité l qu'on appelle *compteur ordinal* est considérée comme adresse propre d'une certaine instruction. On trouve l'ordre qui contient cette adresse propre. L'instruction qui en fait partie est attribuée comme valeur à une quantité r qu'on appelle *registre d'instruction*. On dégage dans r le code d'opération qui détermine le format de l'instruction et la collection d'opérations à exécuter selon cette instruction. L'exécution de l'instruction terminée, le cycle décrit se répète (c'est bien le cycle de fonctionnement du calculateur évoqué au p. 1.1.4). La diversité des calculateurs ne permet pas de donner une description plus détaillée de l'algorithme d'exécution d'un programme. Quelques exemples qu'on trouvera plus bas donnent les algorithmes d'exécution de programmes pour des machines concrètes (v. exemples 2.28, 3.4, 3.6). Dans ces exemples, les actions exécutées après la reconnaissance du code d'opération d'une instruction sont décrites sous forme d'opérateurs dans le YALS.

Exemple 2.28. Décrivons les instructions d'un calculateur à trois adresses que nous appelons C-I. Supposons que l'unité de commande de ce calculateur bien simple est constituée par : 1) un registre d'instruction r; 2) un compteur ordinal l; 3) une cellule logique ω à une position; 4) une cellule logique φ à une position qui sert à contrôler si les résultats d'opérations élémentaires de machine sont proches des résultats d'opérations élémentaires mathématiques respectives.

La cellule ω sert à y inscrire les résultats d'opérations logiques. La cellule φ est souvent appelée registre de dépassement de capacité de l'additionneur (en abrégé, registre de dépassement), et le contenu de ce registre, signal de dépassement, puisque, pour les opérations arithmétiques de machine, l'imprécision du résultat est liée à l'insuffisance du nombre de positions de l'additionneur.

Nous posons que, lorsque la cellule φ se trouve positionnée en 1 (un dépassement se produit), le contenu des registres r, l, ω est rangé respectivement dans les cases de mémoire de numéros 0001, 0002, 0003 et la commande passe à la case de numéro 7777 (où se trouve l'instruction initiale d'un sous-programme spécial qui détermine les actions de la machine dans le cas d'un dépassement de capacité) après quoi la cellule φ est automatiquement positionnée en 0. Lorsque l'état de la cellule φ est 0, le cycle de fonctionnement du calculateur se poursuit normalement. Donc pour $\varphi = 1$ il y a rupture de séquence.

Supposons que les instructions de notre calculateur sont de la forme θabc, où θ est le code d'opération et a,b,c les trois adresses. Alors le registre d'instruction r doit réunir quatre cellules: σ (pour garder θ), i (pour garder a), j (pour garder b) et k (pour garder c). Soient $z_1, z_2, \ldots, z_\nu$ les cases de la mémoire principale du calculateur.

Convenons qu'une instruction de C-I a le format

$$\underbrace{\langle\text{code d'opération}\rangle}_{6\text{ lettres}}\ \underbrace{\langle\text{adresse}\rangle}_{12\text{ lettres}}\ \underbrace{\langle\text{adresse}\rangle}_{12\text{ lettres}}\ \underbrace{\langle\text{adresse}\rangle}_{12\text{ lettres}}$$

En écrivant les instructions de C-I sur des formulaires nous allons utiliser les chiffres octaux. Notons que l'emploi du système octal pour l'écriture des codes binaires n'est pas obligatoire. On se sert parfois du système hexadécimal, et même du système décimal (ceci dépend de la construction des perforateurs).

Donnons un fragment du code d'instructions de C-I (table 2.8) en nous servant du langage des schémas logiques et en désignant par $\varepsilon(z, x, y, \theta)$ le prédicat « le résultat d'une opération mathématique qui correspond à l'opération de machine θ sur les quantités x et y, s'écarte de la quantité z ». En abrégeant, nous écrirons $\varepsilon(z, \theta)$. La rupture de séquence ne sera que mentionnée.

En plus des instructions rangées dans la mémoire principale et appelées par l'unité de commande, on prévoit le déclenchement de certaines opérations à partir du pupitre de commande. Ce peuvent être les instructions:

a) de mise en marche initiale (pression du bouton-poussoir « mise en marche initiale »):

$$\varphi := 0;$$
$$\omega := 0;$$
$$l := 0001;\quad r := z_l;$$

b) de mise en marche (pression du bouton-poussoir « mise en marche »):

$$\varphi := 0;$$
$$l := t;\quad r := z_l;$$

où t est un jeu de tumblers sur le pupitre de commande qui servent à composer le numéro d'une case quelconque de mémoire principale.

Nous supposons que l'arrêt du fonctionnement de la machine est réalisé par programme, par exemple moyennant l'instruction

$$\nu)\ 10\,0000\ \nu\ 0000$$

qui appelle constamment cette même instruction qui sera exécutée sinon indéfiniment, du moins jusqu'à l'intervention de l'opérateur humain qui composera sur le registre de tumblers le numéro, diffé-

Table 2.8

Fragment du code d'instructions de C-I

Code d'opération	Instruction	Description de l'instruction sous forme d'opérateur dans le YALS
01	Addition	$l := l+1$; $z_k := z_l + z_j$; $\omega := z_k < 0$; $\varphi := \varepsilon(z_k, 01)$; $l := (1-\varphi)\cdot l + \varphi\cdot 7777$; $r := z_l$;
02	Soustraction	$l := l+1$: $z_k := z_l - z_j$; $\omega := z_k < 0$; $\varphi := \varepsilon(z_k, 02)$; $l := (1-\varphi)\cdot l + \varphi\cdot 7777$; $r := z_l$;
03	Soustraction des valeurs absolues	$l := l+1$; $z_k := \vert z_l \vert - \vert z_j \vert$; $\omega := z_k < 0$; $\varphi := 0$; $r := z_l$;
04	Multiplication	$l := l+1$; $z_k := z_l \cdot z_j$; $\omega := \vert z_k \vert \geqslant 1$; $\varphi := \varepsilon(z_k, 04)$; $l := (1-\varphi)\cdot l + \varphi\cdot 7777$; $r := z_l$;
05	Division	$l := l+1$; $z_k := z_l : z_j$; $\omega := \vert z_k \vert \geqslant 1$; $\varphi := \varepsilon(z_k, 05)$; $l := (1-\varphi)\cdot l + \varphi\cdot 7777$; $r := z_l$;
06	Transfert	$l := l+1$; $z_k := z_l$; $\omega := z_k < 0$; $\varphi := 0$; $r := z_l$;
07	Extraction de la racine carrée	$l := l+1$; $z_k := \sqrt{z_l}$; $\omega := \vert z_k \vert \geqslant 1$; $\varphi := \varepsilon(z_k, 07)$; $l := (1-\varphi)\cdot l + \varphi\cdot 7777$; $r := r_l$;
10	Branchement inconditionnel	$l := j$; $r := z_l$;
11	Branchement conditionnel	$l := \omega\cdot i + (1-\omega)\cdot j$; $r := z_l$;
12	Addition des instructions	$l := l+1$; $z_k := z_l \boxplus z_j$; $r := z_l$;

rent de v, d'une case de mémoire et pressera le bouton-poussoir « mise en marche ».

L'arrêt peut être réalisé d'une manière plus compliquée ; on peut par exemple composer préalablement sur le pupitre de commande ou bien y envoyer, par un sous-programme spécial, l'instruction « arrêt » et puis procéder comme ci-dessus. Nous omettrons de mentionner dans la table 2.8 les instructions que ce sous-programme devrait utiliser.

EXEMPLE 2.29 Dans les calculateurs modernes, on a déjà dit, le nombre d'adresses et le format d'instructions varient. Considérons, par exemple, le calculateur IBM-360. Le calculateur peut traiter les mots machine de longueur différente. La mémoire rapide est partagée en cases de huit chiffres chacune appelées octets. Tous les octets de la mémoire rapide sont numérotés par des nombres entiers successifs à commencer par 0. La partie adresse contient 24 positions binaires et permet d'adresser les 16777216 (nombre décimal) octets de mémoire rapide. On peut réunir deux cases successives pour ranger un demi-mot (2 octets). Deux demi-mots forment un mot (4 octets). Deux mots forment un double-mot (8 octets). Un groupe d'octets successifs où l'on range un élément d'information s'appelle champ ou zone. Ainsi, ce champ peut avoir la longueur d'un octet, d'un demi-mot, d'un mot, d'un double-mot. On considère comme adresse d'un mot le numéro de son premier octet.

Pour IBM-360, on convient de définir l'adresse d'un demi-mot par un nombre pair, l'adresse d'un mot par un multiple de quatre, l'adresse d'un double-mot par un multiple de huit. En plus des champs de ce type (qui ont des longueurs différentes mais fixes), on admet des champs de longueur variable qui peuvent être adressés par un octet quelconque.

Ainsi, un « mot » dans la machine IBM-360 est déterminé par son octet initial. Les instructions de cette machine sont des demi-mots, des mots, ou se composent de trois demi-mots. Les éléments d'une instruction peuvent se rapporter au premier ou au deuxième opérande. Il existe en tout cinq formats d'instructions que nous donnons ci-dessous en indiquant le numéro de l'opérande auquel se rapporte tel ou tel élément du code de l'instruction.

1. Format du type « registre-registre ». Une instruction de ce format est un demi-mot.

$$\underbrace{\langle\text{Code d'opération}\rangle}_{1\ \text{octet}}\underbrace{\langle\text{registre 1}\rangle}_{0{,}5\ \text{octet}}\underbrace{\langle\text{registre 2}\rangle}_{0{,}5\ \text{octet}}$$

Ce format est propre à une famille d'instructions dont les opérandes sont les contenus de deux registres (leurs adresses ont une longueur de 0,5 octet).

2. Format du type « registre-mémoire » (avec indexation). Une instruction de ce format est un mot.

$$\underbrace{\langle\text{Code d'opération}\rangle}_{1\ \text{octet}}\ \underbrace{\langle\text{registre }1\rangle}_{0{,}5\ \text{octet}}\ \underbrace{\langle\text{index }2\rangle}_{0{,}5\ \text{octet}}\ \underbrace{\langle\text{base }2\rangle}_{0{,}5\ \text{octet}}\ \underbrace{\langle\text{déplacement }2\rangle}_{1{,}5\ \text{octet}}$$

Ce format détermine une famille d'instructions dont les opérandes sont le contenu d'un registre et celui d'une case de mémoire dont l'adresse est la somme du nombre binaire ⟨déplacement 2⟩ et des contenus des registres dont les numéros sont désignés par ⟨index 2⟩ et ⟨base 2⟩.

3. Format du type « registre-mémoire » (sans indexation). Une instruction de ce format est un mot.

$$\underbrace{\langle\text{Code d'opération}\rangle}_{1\ \text{octet}}\ \underbrace{\langle\text{registre }1\rangle}_{0{,}5\ \text{octet}}\ \underbrace{\langle\text{registre }3\rangle}_{0{,}5\ \text{octet}}\ \underbrace{\langle\text{base }2\rangle}_{0{,}5\ \text{octet}}\ \underbrace{\langle\text{déplacement }2\rangle}_{1{,}5\ \text{octet}}$$

Ce format détermine une famille d'instructions à trois opérandes placés dans deux registres (1er et 3e opérandes) et une case dont on obtient l'adresse en additionnant le nombre binaire ⟨déplacement 2⟩ et le contenu du registre ⟨base 2⟩.

4. Format du type « mémoire-instruction ». Une instruction de ce format est un mot.

$$\underbrace{\langle\text{Code d'opération}\rangle}_{1\ \text{octet}}\ \underbrace{\langle\text{opérande }2\rangle}_{1\ \text{octet}}\ \underbrace{\langle\text{base }1\rangle}_{0{,}5\ \text{octet}}\ \underbrace{\langle\text{déplacement }1\rangle}_{1{,}5\ \text{octet}}$$

Ce format détermine une famille d'instructions à deux opérandes; l'adresse du premier est la somme du nombre binaire ⟨déplacement 1⟩ et du contenu du registre adressé comme ⟨base 1⟩, le deuxième opérande figure directement dans le code de l'instruction.

5. Format du type « mémoire-mémoire ». Une instruction de ce format se compose de trois demi-mots.

$$\underbrace{\langle\text{Code d'opération}\rangle}_{1\ \text{octet}}\ \underbrace{\langle\text{longueur }1\rangle}_{0{,}5\ \text{octet}}\ \underbrace{\langle\text{longueur }2\rangle}_{0{,}5\ \text{octet}}\ \underbrace{\langle\text{base }1\rangle}_{0{,}5\ \text{octet}}$$

$$\underbrace{\langle\text{déplacement }1\rangle}_{1{,}5\ \text{octet}}\ \underbrace{\langle\text{base }2\rangle}_{0{,}5\ \text{octet}}\ \underbrace{\langle\text{déplacement }2\rangle}_{1{,}5\ \text{octet}}$$

Une instruction de ce format appartient à une famille d'instructions à deux opérandes dont les longueurs sont indiquées dans le code de l'instruction; les adresses de leurs octets initiaux s'obtiennent comme somme (pour $i = 1, 2$) du nombre binaire ⟨déplacement i⟩ et du contenu du registre adressé comme ⟨base i⟩.

Notons que nous appelons opérandes les données initiales des opérations ainsi que leurs résultats. Les instructions (sauf celles du format 4) contiennent habituellement non pas les opérandes mais leurs apresses.

Remarquons également que, dans le code d'instructions étudié, les adresses de cases de mémoire ne sont pas indiquées directement dans les instructions mais s'obtiennent au cours du décodage de l'instruction.

CHAPITRE 3

PROGRAMMATION

§ 3.1. Etapes principales de la résolution d'un problème au moyen d'un calculateur électronique

On utilise les calculateurs électroniques soit pour résoudre des problèmes isolés (appartenant à une certaine classe), soit pour traiter un groupe de problèmes liés entre eux et appartenant à des classes différentes.

La résolution d'un problème (d'une classe de problèmes) sur calculateur demande plusieurs étapes :

— position mathématique du problème ;

— élaboration d'une méthode de résolution ;

— élaboration d'un algorithme de résolution, rédaction de cet algorithme dans un langage de programmation ;

— programmation ;

— mise au point du programme sur le calculateur ;

— résolution automatique du problème.

Les mêmes étapes font partie d'un travail de rédaction d'un programme complexe pour la résolution d'un groupe de problèmes liés entre eux. De plus, une série d'étapes supplémentaires seront nécessaires pour bâtir à partir de programmes isolés un programme complexe répondant au problème posé. Toutes les questions concernant les systèmes de programmation seront traitées au chapitre 5.

3.1.1. Position mathématique du problème. La tâche consiste à préciser l'ensemble et la nature des données initiales et des résultats cherchés du problème, à représenter le problème en convention mathématique.

Il faut également indiquer comment les résultats dépendent des données initiales, si cette dépendance est connue.

Le formalisme mathématique utilisé à cette étape varie selon la classe des problèmes traités (ce seront par exemple les équations différentielles ordinaires ou les systèmes d'équations s'il s'agit de problèmes de la mécanique ; les équations différentielles à dérivées partielles pour des problèmes de la dynamique du gaz ou de la théorie de l'élasticité ; les systèmes d'équations linéaires algébriques dans le cas de certains problèmes économiques, etc.).

3.1.2. Elaboration d'une méthode de résolution du problème. Une méthode de résolution du problème est considérée comme élaborée du moment que l'on sait comment les résultats cherchés dépendent des données du problème et qu'on trouve les procédés de résolution réalisables sur le calculateur. Dans certains cas on ne peut juger de la validité des méthodes choisies qu'à des étapes suivantes : au stade de l'élaboration de l'algorithme de résolution ou même au cours du traitement du problème sur le calculateur. Si l'on découvre que la méthode n'est pas bonne, on est obligé de reprendre le travail.

3.1.3. Elaboration d'un algorithme de résolution. On élabore un algorithme de résolution d'un problème en se basant sur la méthode choisie à l'étape précédente. Nous savons déjà (v. § 1.5) que la notion d'algorithme est étroitement liée à celle de langage. On établit l'algorithme de résolution dans le langage des descriptions mathématiques, ensuite on l'écrit dans un langage de programmation symbolique (v. § 3.2). Notons qu'il existe des langages de programmation qui ne sont pas symboliques. Une description du processus de résolution du problème faite dans un langage non symbolique n'est pas algorithmique mais elle permet tout de même (à l'étape suivante) d'aboutir à un programme. Les langages symboliques sont plus répandus. Les algorithmes sont généralement écrits dans un des langages suivants : YALS (chap. 2), ALGOL (chap. 7), FORTRAN (chap. 8), COBOL (chap. 10), PL/1 (chap. 9), etc. En élaborant l'algorithme de résolution d'un problème on doit tenir compte des particularités du calculateur utilisé. La rédaction de l'algorithme dans un langage symbolique fait déjà partie de l'étape de programmation.

3.1.3.1. Estimation des particularités du calculateur. En utilisant un calculateur pour la résolution d'un problème il faut tenir compte de ses particularités :

— un nombre suffisamment grand, bien que limité, de chiffres dans la représentation des informations ;

— une vitesse élevée d'exécution des opérations sur les nombres gardés dans la mémoire principale ;

— une vitesse relativement petite d'entrée des données initiales et de sortie des résultats ;

— la capacité de la mémoire principale est relativement réduite, tandis que celle des mémoires auxiliares est très grande ;

— l'éventualité d'erreurs de fonctionnement du calculateur, d'où la nécessité du contrôle.

Une petite vitesse d'échange entre la mémoire principale et les mémoires auxiliaires et une grande vitesse d'opération poussent à réduire le poids des opérations d'échange. D'où l'intérêt d'avoir un programme le plus « court » possible (du point de vue du nombre

d'instructions) qui fournirait le nombre le moindre possible de résultats intermédiaires.

On peut réaliser un grand nombre d'actions avec un programme assez court dans le seul cas où le programme est cyclique, c'est-à-dire qu'il s'exécute plusieurs fois avec des modifications éventuelles. La quantité de nombres qu'il faut conserver en passant d'une opération prévue par l'algorithme de solution à une autre, caractérise la propriété de l'algorithme qu'on appelle connexité. Ainsi, un bon algorithme ne doit pas posséder une grande connexité.

Les données d'un problème à résoudre sur un calculateur contiennent toujours des indications sur la précision voulue. Généralement, on donne la valeur de l'erreur maximale admissible. Seul est acceptable un algorithme qui permet de résoudre le problème avec une erreur ne dépassant pas l'erreur maximale admissible.

Un grand nombre de chiffres dans les représentations des nombres dans une machine assure une grande précision des calculs. Si tout de même les erreurs de calcul s'avèrent trop élevées, la grande vitesse d'opération du calculateur permet parfois d'améliorer la précision en faisant augmenter la quantité d'opérations de machine. Pour les méthodes itératives, on peut prévoir plus d'itérations, et pour les méthodes non itératives, on peut admettre les calculs sur des nombres contenant plus de chiffres qu'il y a de positions dans une case de mémoire. Ceci nécessitera pourtant un plus grand nombre de cases de mémoire pour garder les données initiales et les résultats intermédiaires, le temps de résolution du problème croissant lui aussi remarquablement.

De plus, il faut tenir compte des équipements périphériques du calculateur (nombre et caractéristiques des disques, tambours et bandes magnétiques, d'organes d'entrée et de sortie) et du régime de travail qu'on voudra leur imposer. La répartition du travail entre les différents unités et organes du calculateur lors de l'exécution de l'algorithme de solution s'appelle gestion du programme. On pense à cela déjà à l'étape d'établissement de l'algorithme de solution.

Notons qu'une mauvaise organisation du programme peut le rendre inexécutable par le calculateur. Cette question est étroitement liée au mode d'exploitation du calculateur (en monoprogrammation, en multiprogrammation, en temps partagé; voir chap. 5).

3.1.3.2. Estimation des frais de résolution et des dépenses du temps. En élaborant un algorithme de solution, il faut tenir compte du coût de tous les travaux liés à la résolution du problème ainsi que des dépenses du temps. Si la solution du problème doit être fournie le plus rapidement possible, alors, pour réduire la dépense totale du temps, on accepte même l'augmentation du temps dit de machine. Dans ce cas on cherche à utiliser l'un des algorithmes déjà

connus, autant simple que possible, afin que le temps de programmation soit minimal.

Si le problème n'est pas urgent, ce sont les raisons d'ordre économique qui deviennent primordiales. Par exemple, si l'algorithme de solution d'un problème doit être exécuté de nombreuses fois, il est plus profitable de faire un bon programme, quitte à perdre sur le temps que cela nécessitera, afin d'économiser autant que possible le temps de machine qui coûte cher. Pour estimer approximativement les frais, choisissons pour unité de coût le salaire d'une journée de travail du chercheur qui élabore l'algorithme de solution. Alors une heure de travail d'un calculateur moyen coûtera à peu près 2 fois plus cher et une heure d'un grand calculateur coûtera 6 fois plus cher. D'ailleurs, ces chiffres ne sont qu'approximatifs. Ils ont une tendance bien nette de décroître avec le perfectionnement des calculateurs.

Lorsqu'il s'agit des programmes liés à la machine *) (v. chap. 5), des programmes de traduction automatique des textes, de gestion des fichiers, de commande automatique de phénomènes physiques (par exemple, des processus technologiques), on ajoute les frais de l'analyse mathématique du problème de l'élaboration de l'algorithme de solution, de programmation et de mise au point des programmes au coût du calculateur. Ensuite on évalue les avantages économiques (rapportés à l'unité du temps) qu'on pense tirer de l'utilisation du calculateur et des programmes et on en soustrait les frais d'exploitation et de résolution (rapportés également à l'unité du temps). Ces chiffres permettront d'évaluer le temps au bout duquel le calculateur avec les programmes sera rentable. Il faut que ce temps soit minimal.

3.1.4. Programmation. La programmation consiste à écrire à la main l'algorithme de résolution d'un problème dans un langage de programmation (par exemple, dans un langage d'assemblage (chap. 6), en ALGOL, FORTRAN, COBOL ou PL/1); le programme ainsi obtenu dans un langage symbolique sera ensuite automatiquement traduit en langage machine.

On appelle traduction une transformation équivalente d'un algorithme donné dans un langage de programmation en un algorithme dans le langage machine. Ce travail est réalisé par un programme spécial appelé traducteur ou compilateur (chap. 5).

Pour que le texte rédigé dans un langage de programmation puisse être traduit en langage machine il doit être introduit dans le calculateur. A cet effet le texte est codé sous forme de perforations sur un ruban de papier ou sur des cartes à l'aide d'un perforateur

*) On désigne souvent par ce terme l'ensemble des programmes facilitant l'utilisation de la machine: aides à l'exploitation, aides à la programmation (y compris les bibliothèques de sous-programmes).

à clavier. En même temps, le texte du programme est imprimé sur une large bande de papier pour le contrôle visuel de la perforation. Du ruban perforé, le programme est introduit dans la machine à l'aide d'un lecteur de ruban.

Lorsqu'on utilise les cartes perforées, le contrôle visuel de la perforation est facilité grâce à l'impression du texte sur le bord de la carte.

3.1.5. Mise au point du programme. La mise au point d'un programme sur le calculateur vise les buts suivants : a) la vérification du programme, b) la détection d'erreurs, c) la correction des erreurs trouvées.

La mise au point d'un programme se fait en trois étapes ; on effectue a) le contrôle syntaxique au cours de la traduction, b) une mise au point autonome des parties du programme ; c) une mise au point de tout le programme traduit en langage machine.

Au cours du contrôle syntaxique on élimine les erreurs formelles commises en écrivant le programme dans le langage de programmation. Sans la correction d'erreurs la traduction est impossible.

Pour une mise au point autonome, le programme est divisé en morceaux relativement autonomes. Pour chaque morceau de programme on vérifie l'exactitude des calculs (vérification arithmétique), les transferts de commande réalisés par les instructions de branchement conditionnel (vérification logique), l'exécution des instructions qui modifient d'autres instructions, etc. Pour la vérification arithmétique on fait des calculs à la main sur des exemples spécialement choisis pour confronter ensuite les résultats obtenus à ceux fournis par la machine. Ensuite le programme est exécuté par morceaux soit dans un régime spécial de fonctionnement du calculateur, soit à l'aide de programmes spéciaux de correction permettant d'interrompre l'exécution du programme corrigé à l'endroit voulu, d'introduire les données initiales à partir du pupitre de commande ou à partir d'un périphérique, de comparer les résultats intermédiaires avec les résultats préparés à l'avance, de vérifier l'exécution des instructions de modification. Dans certains systèmes de programmation, on incorpore dans le programme des blocs spéciaux de correction qui sont éliminés au cours du travail du compilateur.

Une fois éliminées les erreurs découvertes lors de la mise au point autonome, on fait une mise au point de l'ensemble du programme. A cet effet on résout quelques exemples spécialement choisis qui font intervenir les différentes parties du programme.

3.1.6. Préparation des données initiales. Résolution du problème au moyen du calculateur. Pour être introduites dans le calculateur les données initiales du problème, qui sont représentées sur des

formulaires doivent être transférées sur un support mécanographique sous forme de perforations. Ce procédé est réalisé à l'aide de perforateurs à clavier.

Toutes les erreurs de perforation et de lecture sont à éliminer.

L'une des méthodes de détection d'erreurs de perforation est ce qu'on appelle vérification. Elle consiste en ce que les cartes ou les rubans perforés sont introduits dans un dispositif spécial à clavier, appelé vérificateur, sur lequel on compose une seconde fois le texte déjà perforé (numérique ou alphanumérique). Si les deux textes ne se confondent pas, le dispositif délivre un signal avertissant l'opérateur humain qu'il faut vérifier le texte et éliminer les erreurs.

Cette méthode de vérification a un défaut : il est impossible de repérer une faute du dispositif de lecture. Il existe des dispositifs qui, au lieu de coder l'information sur un support mécanographique, l'enregistrent sur une bande magnétique.

L'enregistrement se fait deux fois, par deux techniciens. A peu près quarante opérateurs peuvent utiliser simultanément un enregistreur.

L'information enregistrée est introduite dans le calculateur où elle subit une vérification par comparaison de deux exemplaires d'un même texte. Lorsque les textes sont différents, une liste d'erreurs est imprimée.

La méthode décrite permet d'économiser sur le matériel mécanographique (cartes et rubans de papier), et c'est là son avantage. Elle exige pourtant de mettre en œuvre un matériel d'enregistrement magnétique coûteux et son service.

Il existe une méthode d'élimination des fautes de perforation et de lecture d'information numérique basée sur l'utilisation du calculateur même et qui ne demande aucun équipement supplémentaire. Elle consiste en ce que l'information numérique est partagée en groupes de longueur relativement petite (par exemple, un groupe peut représenter une ligne d'une table) ; on additionne les nombres de chaque groupe en marquant la somme à la fin du groupe. Ceci fait, on effectue la perforation de l'information, les sommes de contrôle y compris, et on introduit l'information dans la machine. Au fur et à mesure de l'entrée de l'information, la machine calcule les sommes des groupes de nombres et les compare avec les sommes préparées à l'avance. L'égalité des sommes permet de conclure à l'absence d'erreurs. Le groupe considéré comme correct est enregistré sur la bande magnétique. Lorsque les sommes comparées diffèrent, le groupe correspondant n'est pas enregistré sur la bande magnétique, et la machine effectue l'impression de la ligne erronée (y compris la somme de contrôle). Une fois l'erreur éliminée l'introduction de l'information se poursuit de la même façon.

Cette dernière méthode demande certaines dépenses de cartes ou rubans de papier qui sont systématiques, bien qu'assez réduites.

De plus, l'opération d'inscription de sommes de contrôle sur des formulaires est très désavantageuse. La sommation se fait au moyen d'automates à clavier et le travail est à peu près équivalent à la perforation.

La méthode se perfectionne dès que l'inscription des sommes de contrôle se fait automatiquement. Pour cela on doit disposer de calculatrices spéciales qui impriment la somme de contrôle à la fin de la ligne des nombres additionnés.

On obtient de bons résultats du point du vue de la vitesse d'entrée et de l'efficacité du contrôle en introduisant l'information directement dans le calculateur au moyen de plusieurs dispositifs à clavier. Ceci permet d'économiser sur le matériel mécanographique. Le texte est deux fois lu par la machine, les deux lectures sont comparées, les mots qui ne coïncident pas sont considérés comme erronés. Cette méthode exige que la machine fonctionne en temps partagé. Les grandes dépenses du temps de machine représentent un défaut de la méthode.

Après l'entrée dans le calculateur des programmes et des données initiales, la résolution du problème se fait automatiquement. Tout de même, l'assistance d'un opérateur est nécessaire; celui-ci intervient lorsqu'une situation inadmissible ou imprévue se produit. L'opérateur agit en conformité des prescriptions fournies par le programmeur qui a rédigé le programme.

§ 3.2. Langages de programmation

On appelle *langage de programmation* un système de symboles utilisé pour la description des procédés de résolution de problèmes au moyen de calculateurs. Par leur nature, les langages de programmation se classent en trois groupes: 1) les langages symboliques (ou algorithmiques) formels; 2) les langages non symboliques formels; 3) les systèmes de symboles non complètement formalisés qu'on utilise dans la programmation.

3.2.1. Langages symboliques formels. Ce groupe contient: a) les langages symboliques des machines et des systèmes d'exploitation; b) les langages symboliques orientés vers la machine; c) les langages symboliques orientés vers les problèmes; d) les langages symboliques universels indépendants de la machine.

Les particularités des langages algorithmiques de machine sont décrites au p. 2.3.9. On appelle langage de système d'exploitation un langage symbolique assimilable à un système formé d'un calculateur et d'un programme appelé superviseur. Du point de vue du

programmeur, ce complexe représente un entier, on dirait un nouveau calculateur différent du calculateur initial qui en fait partie.

A l'heure actuelle on ne fait plus de programmation manuelle dans le langage machine parce que ce genre du travail exige que le programmeur retienne un grand nombre de détails, souvent embrouillés, d'où les erreurs de programmation inévitables. On programme parfois à la main à l'étape initiale d'élaboration d'un système d'exploitation (§ 5.4), lorsqu'il est impossible pour une raison ou une autre de se servir d'un autre calculateur déjà muni de programmes nécessaires. Et pourtant, toute programmation a pour but de fournir un programme dans le langage machine, quelles que soient les méthodes de programmation

Les langages orientés vers la machine possèdent des moyens d'expression permettant, dans la notation de l'algorithme, d'indiquer par quels moyens techniques on doit réaliser telle ou telle partie de l'algorithme et comment utiliser la mémoire. Le groupe des langages orientés vers la machine comprend les autocodes dont les possibilités les rapprochent des langages machine (tel est par exemple le langage d'assemblage décrit au chapitre 6), ainsi que certains langages se rapprochant des langages symboliques universels, par exemple ALMO (voir [15]).

Dans le groupe des langages symboliques orientés vers les problèmes sont rassemblés les langages spécialement conçus pour la description des procédés de résolution de problèmes d'une classe restreinte, par exemple, de l'algèbre linéaire, de la statistique, de la programmation linéaire, du traitement des fichiers, etc. En particulier, le COBOL (chap. 10) est un langage de ce type.

Les langages universels indépendants de la machine servent à créer les algorithmes de résolution des problèmes les plus divers. Ce sont, par exemple, les langages déjà cités ALGOL, FORTRAN, PL/1. Les particularités du calculateur sont prises en considération dans le compilateur.

Le langage YALS représente une exception dans le groupe des langages symboliques universels indépendants de la machine. Ce n'est pas seulement un langage de programmation. Or, en tant que tel, il est utilisé non pas à la rédaction d'un programme-source qui sera ensuite traduit dans le langage machine, mais à la première étape de description des algorithmes, lorsqu'on programme dans un langage machine ou dans un langage d'assemblage (méthode de programmation opératorielle, voir § 3.4). Un algorithme rédigé dans le YALS est traduit à la main dans le langage machine ou dans le langage d'assemblage. Dans le dernier cas, une traduction du langage d'assemblage dans le langage machine, s'impose.

Dans la table 3.1. ci-dessous sont rassemblées quelques caractéristiques des langages symboliques.

Table 3.1

Comparaison des langages de programmation symboliques

Classe de langages	Prise en considération des particularités de la machine	Classe de problèmes	Méthode de programmation	Estimation conventionnelle de la qualité des programmes
Langages de machine	complète	déterminée par la machine	manuelle	haute
Langages orientés vers la machine	partielle	déterminée par la machine	automatisée	satisfaisante
Langages orientés vers les problèmes	faible	étroite	automatisée	satisfaisante ou basse
Langages universels indépendants	nulle ou très faible	très vaste	automatisée	basse

3.2.2. Langages de programmation non symboliques formels. A ce groupe se rapportent les langages permettant de décrire la méthode de résolution d'un problème sans indiquer la succession exacte des actions à accomplir. Citons en guise d'exemple le langage des schémas paramétriques, où le processus de traitement de l'information est donné sous forme d'un ensemble de couples d'opérations, la première opération du couple représentant une condition, la seconde, une action de traitement de l'information. Seules seront exécutées les actions qui correspondent aux conditions satisfaites. Tant qu'une condition n'est pas satisfaite, on n'accomplit pas l'action qui lui répond.

Si plusieurs conditions sont satisfaites simultanément, on exécute les actions correspondantes soit simultanément, soit dans un ordre arbitraire. De tels langages sont utilisés aussi bien dans la programmation automatisée que manuelle.

3.2.3. Systèmes de symboles non complètement formalisés. Ces langages de programmation sont habituellement utilisés soit lors d'une programmation manuelle, soit à l'étape préliminaire, manuelle elle aussi, de programmation automatisée. En tant qu'exemples, signalons les *organigrammes*, ou les diagrammes de déroulement du traitement, et les *désignations non formelles* utilisées par le programmeur. Un organigramme d'un programme représente une description plus ou moins détaillée du programme, où les différentes parties du programme sont représentées comme « blocs » (rectangles,

losanges, cercles, etc.), à l'intérieur desquels on décrit, dans une langue naturelle (par exemple, en français), le contenu de la partie correspondante. Les liaisons entre les blocs (les parties du programme) sont figurées par des lignes qui désignent des transferts de la commande. Une ligne peut être munie d'une inscription indiquant les conditions pour lesquelles le passage de commande correspondant est réalisé.

Les organigrammes sont analogues aux algorithmes donnés en YALS au moyen d'opérateurs généralisés, mais en diffèrent par le fait que la signification des blocs est décrite dans une langue naturelle, non formelle, tandis que dans le YALS les opérateurs généralisés sont munis d'un décodage en langage formel.

Quant aux désignations non formelles [5], on s'en servait à l'époque de la programmation manuelle où les méthodes plus perfectionnées de programmation n'étaient pas encore inventées. En élaborant un programme, on écrivait les instructions dans le langage machine et, parallèlement, on décrivait les opérations à exécuter dans un langage « personnel » des descriptions mathématiques (c'est-à-dire que chaque programmeur utilisait ou même inventait son langage à lui). L'ambiguïté des désignations non formelles en a fait très vite un langage désuet, tout à fait inutilisable au niveau élevé de la programmation moderne.

§ 3.3. Programmation en langage machine

Bien que, actuellement, on ne programme plus directement dans le langage machine, ni avec les moyens intermédiaires, tels que la notation préalable de l'algorithme en YALS ou sa représentation sous forme d'organigramme, nous nous arrêtons quand même sur ce type de programmation et ceci pour deux raisons : premièrement, on comprend mieux les principes d'action des calculateurs, deuxièmement, c'est intéressant du point de vue historique puisque cela donne l'idée du développement des méthodes de programmation

3.3.1. Liaison entre la répartition de la mémoire et la composition des instructions. A l'époque des premiers calculateurs commandés par programmes, dont les possibilités étaient encore assez réduites, il n'existait pas de méthodes de programmation. La rédaction de programmes était considérée comme une sorte de problème combinatoire. Mais on a tôt compris que la programmation réunit deux genres du travail : la répartition de la mémoire de la machine et la composition des instructions. La répartition de la mémoire, i.e. le rangement dans la mémoire de la machine des données et du programme du problème (données initiales, instructions, codes auxiliaires, résultats intermédiaires et définitifs) est étroitement liée à la composition des instructions. On ne peut pas écrire les

instructions d'un programme sans connaître les numéros des cases de mémoire où sont rangées les données initiales et les codes auxiliaires, sans décider où placer les résultats. D'autre part, il est difficile de disposer le matériel dans la mémoire de la machine sans connaître à l'avance le nombre d'instructions du programme et celui des résultats intermédiaires que l'on doit garder simultanément dans la mémoire.

3.3.2. Adresses symboliques. Le premier perfectionnement du processus de programmation fut la symbolisation d'adresses. Cette méthode a permis de séparer ces deux genres du travail.

On divise la suite M de toutes les cases de la mémoire rapide en zones (sous-suites) M_1, M_2, . . ., M_i, . . . On désigne les cases de la zone M_i par $\alpha_i + 1$, $\alpha_i + 2$, $\alpha_i + 3$, . . ., où α_i $(i = 1, 2, \ldots)$ sont des lettres d'un alphabet quelconque. Les codes $\alpha_i + 1$, $\alpha_i + 2$, . . . $(i = 1, 2, \ldots)$ s'appellent *adresses symboliques* ou *désignations alphanumériques des cases*. (On peut poser, par exemple, que les données initiales sont gardées dans les cases $b + 1$, $b + 2$, . . ., les résultats intermédiaires, dans les cases $c + 1$, $c + 2$, . . . , le programme, dans les cases $a + 1$, $a + 2$, . . ., etc.)

En se servant des adresses symboliques, on rédige le programme directement d'après les formules de l'algorithme de résolution. Le programme ainsi obtenu contient les adresses symboliques au lieu des adresses réelles. Il est facile maintenant de déterminer le nombre de cases à réserver dans les zones M_1, M_2, . . ., M_i, . . . On répartit la mémoire en attribuant aux lettres α_1, α_2, . . ., α_i, . . . certaines valeurs numériques. Une forme définitive du programme s'obtient en remplaçant les symboles dans les adresses des instructions par les valeurs qui leur sont attribuées. Le dernier procédé, de caractère « méchanique », s'appelle *attribution d'adresses réelles*.

L'emploi des adresses symboliques a les avantages suivants:

1. La rédaction du programme et la répartition de la mémoire se font séparément ce qui facilite sensiblement la tâche du programmeur.

2. Il est possible de répartir avantageusement le travail entre un programmeur qualifié (qui rédige le programme en adresses symboliques et distribue la mémoire) et un groupe de programmeurs moins qualifiés qui sont chargés de l'attribution d'adresses réelles.

3. Il est plus facile de rédiger les instructions de modification qui se rencontrent assez souvent dans des programmes.

4. Le procédé d'attribution d'adresses réelles peut être automatisé, c'est-à-dire le calculateur peut l'effectuer tout seul d'après un programme composé une fois pour toutes.

EXEMPLE 3.1. Rédiger un programme de résolution sur la machine C-I (voir exemple 2.28) du système des équations linéaires

$$Ax + By = C,$$
$$Dx + Ey = F,$$

qui satisfait à la condition

$$AE - BD \neq 0.$$

L'algorithme de résolution se réduit aux calculs d'après les formules

$$x = \frac{CE - BF}{AE - BD},$$
$$y = \frac{AF - CD}{AE - BD}.$$

Supposons que les nombres sont rangés dans la mémoire de la manière suivante:

$$b + 1)\ A;\quad b + 2)\ B;\quad b + 3)\ C;$$
$$b + 4)\ D;\quad b + 5)\ E;\quad b + 6)\ F.$$

Ici, à gauche de la parenthèse qui se ferme, se trouve le numéro symbolique de la case, et à droite, le contenu de cette case.

Réservons les cases de numéros $a + 1$, $a + 2$, . . . aux instructions du programme; les cases $c + 1$, $c + 2$, $c + 3$ aux résultats intermédiaires et les cases $d + 1$ et $d + 2$ aux résultats définitifs.

Le programme du calcul de x et y en adresses symboliques aura la forme

$a + 1)$	04	$b + 1$	$b + 5$	$c + 1$	calcul de AE,
$a + 2)$	04	$b + 2$	$b + 4$	$c + 2$	calcul de BD,
$a + 3)$	02	$c + 1$	$c + 2$	$c + 1$	calcul de $AE - BD$,
$a + 4)$	04	$b + 3$	$b + 5$	$c + 2$	calcul de CE,
$a + 5)$	04	$b + 2$	$b + 6$	$c + 3$	calcul de BF,
$a + 6)$	02	$c + 2$	$c + 3$	$c + 2$	calcul de $CE - BF$,
$a + 7)$	05	$c + 2$	$c + 1$	$d + 1$	calcul de x,
$a + 10)$	04	$b + 1$	$b + 6$	$c + 2$	calcul de AF,
$a + 11)$	04	$b + 3$	$b + 4$	$c + 3$	calcul de CD,
$a + 12)$	02	$c + 2$	$c + 3$	$c + 2$	calcul de $AF - CD$,
$a + 13)$	05	$c + 2$	$c + 1$	$d + 2$	calcul de y,
$a + 14)$	10	0000	$a + 14$	0000	arrêt.

Le « morceau » donné de programme ne contient pas d'instructions d'entrée de l'information, de sa traduction en langage machine interne, de conversion des résultats dans le système décimal usuel et d'impression. Ces instructions sont omises pour simplifier l'exemple.

Rappelons que dans notre cas, les numéros des cases sont notés sur des formulaires en numération octale.

Faisons l'attribution d'adresses réelles. Supposons que la première instruction de notre « morceau » de programme sera rangée à la case numéro 0031. Il faut donc poser

$$a = 0030.$$

Comme la longueur de la partie considérée du programme est 14 (en numération de base huit), on peut poser

$$b = a + \Delta a, \text{ où } \Delta a > 14.$$

Par exemple, en posant $\Delta a = 100$ nous aurons

$$b = 0030 + 0100 = 0130.$$

D'une façon analogue, nous aurons

$$c = b + 6 = 0136,$$
$$d = c + 3 = 0141.$$

Remplaçons les lettres dans le programme par les valeurs d'affectation. Le programme prendra la forme:

0031)	04	0131	0135	0137
0032)	04	0132	0134	0140
0033)	02	0137	0140	0137
0034)	04	0133	0135	0140
0035)	04	0132	0136	0141
0036)	02	0140	0141	0140
0037)	05	0140	0137	0142
0040)	04	0131	0136	0140
0041)	04	0133	0134	0141
0042)	02	0140	0141	0140
0043)	05	0140	0137	0143
0044)	10	0000	0044	0000

. . . .

0131) . . . A
0132) . . . B
0133) . . . C
0134) . . . D
0135) . . . E
0136) . . . F

0137) ⎫
0140) ⎬ résultats intermédiaires
0141) ⎭
0142) . . . x
0143) . . . y

Très fructueuse, la méthode d'adresses symboliques illustrée par cet exemple est utilisée dans les méthodes plus évoluées.

§ 3.4. Méthode de programmation opératorielle

La méthode opératorielle apparut comme un perfectionnement de la programmation manuelle. Elle marqua une nouvelle étape du développement des méthodes de programmation. C'est la méthode opératorielle qui conduisit aux premiers résultats efficaces dans le domaine de la programmation automatique. En 1953 A. Liapounov formula la notion d'opérateur dans la programmation, et en 1954 l'Institut des Mathématiques de l'Académie des sciences de l'U.R.S.S. élabora le programme de programmation automatique PP-1 qui fut le premier compilateur d'un langage algorithmique universel dans le langage machine.

En tant que méthode de programmation manuelle, la méthode opératorielle n'a pas perdu sa valeur jusqu'à nos jours, parce qu'elle reste un moyen efficace de programmation dans le langage d'assemblage. Il est encore à souligner que la forme actuelle de la méthode opératorielle diffère radicalement de ce qu'elle représentait en 1953.

Les schémas logiques de l'époque et le langage source du programme PP-1 se sont transformés en le langage algorithmique, le YALS, et les procédés presque intuitifs de transformation d'un schéma logique en un programme ont pris le caractère d'une série de transformations équivalentes ordonnées de l'algorithme (chap. 4).

3.4.1. L'essentiel de la méthode de programmation opératorielle. Citons les traits essentiels de la méthode de programmation opératorielle.

1. L'algorithme de résolution du problème est donné dans le langage algorithmique des schémas logiques (YALS).

2. L'algorithme initial peut subir une transformation équivalente pour obtenir un nouvel algorithme dans le YALS, mieux adapté à la traduction en langage machine. Bien que la programmation se fasse dans le langage des schémas logiques, on tient néanmoins compte des propriétés de la machine pour laquelle on programme, et plus particulièrement, du code d'instructions.

3. On effectue une transformation équivalente des opérateurs de l'algorithme donné en YALS, qui consiste à représenter les opérateurs élémentaires sous forme d'une suite d'opérateurs élémentaires

du type utilisé pour la description des instructions de la machine. En même temps, si c'est possible, on groupe les opérateurs élémentaires de la même façon qu'ils sont groupés dans les descriptions des instructions de la machine.

4. Chaque opérateur élémentaire ou chaque groupe d'opérateurs élémentaires est remplacé par l'instruction correspondante de la machine. En même temps on met en correspondance aux cellules du langage des schémas logiques les cases de la mémoire du calculateur.

Le groupe des ordres du programme qui correspond à un opérateur de l'algorithme en YALS s'appelle *opérateur du programme*. Ainsi, selon la méthode opératorielle, l'algorithme donné en YALS subit d'abord une transformation équivalente dans son ensemble, ensuite les transformations portent sur les opérateurs sans modifier le schéma logique et, enfin, vient l'étape d'inscription dans le langage machine.

Dans la programmation manuelle, on effectue les transformations équivalentes mentionnées de manière non formelle, bien que cela conduise parfois à des erreurs. Et pourtant, dans la plupart des cas, les transformations nécessaires sont très simples, de sorte que le programmeur les effectue sans peine « mentalement ».

L'un des avantages du YALS est de permettre l'écriture d'algorithmes au niveau abstrait, ou même au niveau des opérateurs généralisés. Ceci facilite la rédaction de l'algorithme et sa transformation.

EXEMPLE 3.2. Dresser la table des valeurs d'une fonction $y=f(x)$ pour $x = x_1, x_2, \ldots, x_n$, si

$$f(x)=\begin{cases}\varphi(x) & \text{pour } x<x^*,\\ \varphi(x)+\theta(x) & \text{pour } x\geqslant x^*,\end{cases}$$

$$x^* = \text{const}.$$

A cette fin on peut se servir de l'algorithme suivant donné dans un français formalisé.

1°. Poser $i = 1$; aborder le p. 2°.

2°. Calculer $z = \varphi(x_i)$. Aborder le p. 3°.

3°. Vérifier la condition $x \geqslant x^*$. Si oui, passer au p. 4°, si non, au p. 5°.

4°. Obtenir la valeur cherchée de la fonction $y_i = z + \theta(x_i)$. Aborder le p. 6°.

5°. Poser $y_i = z$. Passer au p. 6°.

6°. Vérifier l'égalité $i = n$. Si oui, passer au p. 8°, si non, au p. 7°.

7°. Augmenter la valeur de i de 1. Revenir au p. 2°.

8°. Fin du processus.

En considérant les points de cet algorithme comme opérateurs on peut écrire :

$$I_0 V_1 \underset{4}{\lrcorner} D_1^i P_1^i \underset{1}{\llcorner} \; D_2^i \underset{2}{\llcorner} \; \underset{1}{\lrcorner} D_3^i \underset{2}{\lrcorner} P_2 \overset{3}{\ulcorner} V_2 \underset{4}{\llcorner}$$

$$\underset{3}{\lrcorner} T\, [V_1 i := 1;\; D_1^i z := \varphi(x_i);\; P_1^i x_i \geqslant x^*;$$

$$D_2^i y_i := z + \theta(x_i);\; D_3^i y_i := z;\; P_2 i = n;\; V_2 i := i + 1;] \qquad (3.1)$$

Laissant de côté les opérations d'entrée-sortie, de traduction dans le langage machine interne, de conversion des résultats, établissons le programme pour C-I par la méthode opératorielle.

Pour fixer les idées, posons

$$\varphi(x) = x^5 + x + 1, \qquad \theta(x) = \sqrt{x} + 2.$$

La première étape de programmation est déjà réalisée : nous avons établi l'algorithme de résolution du problème en YALS.

A la deuxième étape nous ferons des transformations équivalentes si sous sa forme originale l'algorithme ne s'adapte pas bien au calculateur donné. Dans notre cas il faut remplacer l'expression $P_1^i \underset{1}{\llcorner}$ par $P_3^i \overset{1}{\ulcorner}$, où P_3^i est $x_i < x^*$, et l'expression $P_2 \overset{3}{\ulcorner}$ par $P_4 \underset{3}{\llcorner}$, où P_4 est $i < n$. Ces transformations ne sont pas des transformations internes des opérateurs, puisqu'elles modifient les signes de passage ouvrants. Leur légitimité est évidente. En effet, $x_i < x^*$ est faux lorsque $x_i \geqslant x^*$ est vrai, et vrai lorsque $x_i \geqslant x_i^*$ est faux. Donc, en remplaçant P_1^i par P_3^i, on doit remplacer le symbole $\underset{1}{\llcorner}$ par le symbole $\overset{1}{\ulcorner}$ (pour ne pas modifier le processus de calcul).

Considérons l'opérateur P_2. Il donne la réponse « non » tant que $i < n$, et fournira la réponse « oui » dès que la condition $i = n$ est satisfaite. La relation $i > n$ ne sera jamais vraie lors de l'exécution de l'algorithme. Ceci vu, on doit remplacer le signe de passage ouvrant respectif par le signe de passage ouvrant opposé, lorsqu'on remplace P_2 par P_4. Les transformations décrites sont imposées par le fait que les instructions de C-I sont mieux adaptées à la vérification de la condition $<$.

La troisième étape de la méthode opératorielle représente les transformations internes des opérateurs dans le but de les rapprocher des descriptions des instructions du calculateur. Le bien-fondé de ces transformations sera éclairci au chapitre 4. Pour le moment, nous nous bornerons à dire qu'elles sont parfaitement admissibles.

Après ces trois étapes, l'algorithme de résolution du problème aura la forme

$$I_0 V_1 \underset{4}{\rfloor} D_1^i P_3^i \overset{1}{\ulcorner} D_2^i \underset{2}{\llcorner} \underset{1}{\rfloor} D_3^i \underset{2}{\rfloor} P_4 \underset{3}{\llcorner} V_2 \underset{4}{\llcorner} \underset{3}{\rfloor} T\, [V_1 i := 1; \quad D_1^i z := x_i \cdot x_i;$$
$$z := z \cdot z; \; z := z \cdot x_i; \; z := z + x_i; \; z := z + 1; \; P_3^i x_i < x^*;$$
$$D_2^i u := \sqrt{x}; \; u := u + 2; \; y_i := z + u; \; D_3^i y_i := z; \; P_4 i < n; \; V_2 i := i + 1;] \qquad (3.2)$$

A la quatrième étape nous nous servirons de la méthode d'adresses symboliques.

Supposons que les données initiales soient disposées dans la mémoire de la façon suivante:

$$b+1)\; x_1; \quad b+2)\; x_2; \quad \ldots; \quad b+n)\; x_n.$$

Les résultats cherchés seront placés dans les cases:

$$d+1)\; y_1; \quad d+2)\; y_2; \quad \ldots; \quad d+n)\; y_n.$$

Les cases $c+1$ et $c+2$ contiendront les résultats intermédiaires. Les constantes seront réparties comme suit:

$e+1)$ 1; $e+2)$ 2; $e+3)$ x^*; $e+4)$ 06 $c+1$ 0000 $d+n$;
$e+5)$ 00 0001 0001 0000; $e+6)$ 00 0001 0000 0000;
$e+7)$ 00 0000 0000 0001.

Comme le code d'instructions de C-I contient l'instruction de l'addition des instructions, alors, dans le cas où les adresses dépendent linéairement de paramètres, on peut ne pas garder dans la mémoire les valeurs de paramètres, mais d'en tenir compte par modification des instructions dépendant de paramètres. En pratique on emploie un mode d'adressage plus économique, celui d'indexation, mais ceci nécessite que le calculateur possède des registres d'index (voir p. 3.5.1). Or, la description de C-I ne contient pas de registres d'index. Revenons donc au premier procédé. Il est évident qu'on n'a pas besoin de représenter dans le programme l'opérateur V_1, il faut seulement écrire les instructions dépendant de paramètres sous la forme correspondant au cas $i = 1$. L'opérateur D_1^i aura la forme (pour $i = 1$)

$a+1)$	04	$b+1$	$b+1$	$c+1$
$a+2)$	04	$c+1$	$c+1$	$c+1$
$a+3)$	04	$b+1$	$c+1$	$c+1$
$a+4)$	01	$b+1$	$c+1$	$c+1$
$a+5)$	01	$c+1$	$e+1$	$c+1$

L'opérateur P_3^i s'écrira (pour $i = 1$)

$a + 6)$ 02 $b + 1$ $e + 3$ $c + 2$

Nous n'avons pas besoin de la différence que nous fournit cette instruction de soustraction et nous l'envoyons dans la case $c + 2$ où elle sera bientôt effacée. Il nous faut seulement la valeur de ω. Le signe de passage $\overset{1}{\ulcorner}$ sera interprété par l'instruction

$a + 7)$ 11 $k + 1$ $a + 10$ 0000

L'opérateur D_2^i et le signe de passage $\underset{2}{\llcorner}$ s'écriront

$a + 10)$	07	$b + 1$	0000	$c + 2$
$a + 11)$	01	$c + 2$	$e + 2$	$c + 2$
$a + 12)$	01	$c + 1$	$c + 2$	$d + 1$
$a + 13)$	10	0000	$l + 1$	0000

L'opérateur D_3^i précédé dans le schéma logique du signe $\underset{1}{\lrcorner}$ aura la forme

$k + 1)$ 06 $c + 1$ 0000 $d + 1$

L'opérateur P précédé du signe $\underset{2}{\lrcorner}$ vérifiera l'inégalité entre l'instruction se trouvant dans la case $k + 1$ et la constante ayant la forme d'instruction et disposée dans la case $e + 4$. Il sera interprété par l'ordre

$l + 1)$ 02 $k + 1$ $e + 4$ $c + 2$

La différence que nous fournit cette instruction nous étant inutile, nous l'enverrons à la case $c + 2$, en utilisant la valeur du signal ω dans l'ordre correspondant au signe de passage $\underset{3}{\llcorner}$:

$l + 2)$ 11 $l + 3$ $m + 1$ 0000

L'opérateur V_2 calculera non pas la nouvelle valeur du paramètre i, mais la nouvelle forme de toutes les instructions dépendant de ce paramètre, c'est-à-dire qu'il modifiera leurs adresses au moyen des constantes gardées dans les cases $e + 5$, $e + 6$, et $e + 7$. Cet opérateur aura la forme

$l + 3)$	12	$a + 1$	$e + 5$	$a + 1$
$l + 4)$	12	$a + 3$	$e + 6$	$a + 3$
$l + 5)$	12	$a + 4$	$e + 6$	$a + 4$
$l + 6)$	12	$a + 6$	$e + 6$	$a + 6$
$l + 7)$	12	$a + 10$	$e + 6$	$a + 10$
$l + 10)$	12	$a + 12$	$e + 7$	$a + 12$
$l + 11)$	12	$k + 1$	$e + 7$	$k + 1$

Enfin, l'opérateur T sera

$$m+1)\quad 11\quad 0000\quad m+1\quad 0000$$

Pour répartir la mémoire, il faut attribuer certaines valeurs aux lettres a, b, c, d, e, k, l, m tout en cherchant à satisfaire les relations

$$k = a + 13,$$
$$l = k + 1,$$
$$m = l + 11.$$

Les relations entre les autres lettres peuvent être quelconques.

3.4.2. Organisation des programmes. On a déjà dit au p. 3.1.3.1. qu'une organisation rationnelle du programme est l'élément important de son élaboration. On sous-entend par organisation d'un programme la répartition des tâches entre les différents dispositifs du calculateur. On doit prévoir dans le programme des portions spéciales qui assurent l'échange d'information entre les différentes mémoires du calculateur. Il apparaît alors une série de nouveaux opérateurs, étroitement liés à la machine, que l'on doit inclure dans l'algorithme.

On considère comme éléments de l'organisation :

— l'entrée et la sortie du programme et des données initiales par chaque dispositif d'entrée ou de sortie ;

— l'enregistrement sur bande magnétique et la lecture (pour chaque dérouleur de bande) ;

— l'enregistrement sur (chaque) tambour magnétique et la lecture ;

— le transfert des codes à l'intérieur de la mémoire rapide ;

— l'appel d'un bloc du programme de l'utilisateur ;

— l'exécution de la partie du programme qui réalise l'algorithme principal ;

— l'arrêt de la machine ;

— l'envoi de signaux au pupitre de commande et la réaction à ces signaux, etc.

Dans la méthode de programmation opératorielle, les opérateurs auxiliaires servant à l'organisation du programme sont inclus dans l'algorithme de résolution du problème et sont traduits ensuite dans le langage machine. On adapte l'algorithme à la machine après la troisième et avant la quatrième étape du travail selon la méthode opératorielle. Les opérations d'organisation du programme nécessitent souvent une correction de l'algorithme de résolution.

La question d'organisation ne se pose pas dans le seul cas où il s'agit des programmes relativement courts et peu compliqués. Néanmoins, avec le développement des calculateurs, le problème de

gestion des programmes se trouve confié au calculateur lui-même. Quelques moyens d'organisation des mémoires auxiliaires sont décrits dans [27].

Exemple 3.3. Pour illustrer tant soit peu ce qu'on a dit sur l'organisation des programmes, revenons à l'exemple 3.2. Supposons que n est si grand que la mémoire principale du calculateur ne peut recevoir que la moitié des données initiales et résultats. Posons pour simplifier que n est pair. On peut résoudre le problème en appliquant d'abord l'algorithme (3.2) aux $n/2$ premières données initiales, en faisant sortir les résultats, puis en répétant le même programme pour les données qui restent. On arrive à cette fin par une organisation rationnelle du programme. Introduisons le paramètre j dont la valeur est le nombre des séries de résultats sorties du calculateur. Dans l'algorithme de résolution obtenu à la troisième étape de programmation par la méthode opératorielle, remplaçons l'opérateur P_4 par l'opérateur P_5 dont la signification est

$$i < n/2.$$

L'algorithme qu'on obtient peut être représenté comme

$$I_0 R(\underset{3}{\llcorner}) \underset{3}{\lrcorner} T \text{ [décodage]},$$

où

$$R(\underset{3}{\llcorner}) = V_1 \underset{4}{\lrcorner} D_1^i P_3^i \overset{1}{\ulcorner} D_2^i \underset{2}{\llcorner} \underset{1}{\lrcorner} D_3^i \underset{2}{\lrcorner} P_5 \underset{3}{\llcorner} V_2 \underset{4}{\llcorner}$$

Introduisons les désignations:

W, l'opérateur d'entrée de $n/2$ nombres et de leur rangement dans les cases $a+1, a+2, \ldots, a+n/2$;

A, l'opérateur de traduction dans le langage machine des nombres introduits et de rangement de résultats dans les mêmes cases;

B, l'opérateur de conversion des $n/2$ résultats en codes alphanumériques (préparation à l'impression);

S, l'impression des $n/2$ résultats;

V_3, l'opérateur $j := 0$;

V_4, l'opérateur $j := j + 1$;

P_6 l'opérateur $j = 2$;

L'algorithme de résolution avec les opérateurs d'organisation du programme aura la forme

$$I_0 V_3 \underset{5}{\lrcorner} WAR(\underset{3}{\llcorner}) \underset{3}{\lrcorner} BSV_4 P_6 \underset{5}{\llcorner} T_1 \text{ [décodage]}.$$

Il est clair qu'après une telle transformation de l'algorithme on ne peut plus rejeter V_1 à l'étape de représentation des opérateurs par les instructions, comme on l'a fait dans l'exemple 3.2, puisque

cette fois l'opérateur V_1 devra restituer toutes les instructions dépendant du paramètre i après la première série de calculs. En cas de C-I, l'opérateur V_1 doit enregistrer dans les cases correspondantes la forme initiale des instructions dépendant de i.

§ 3.5. Quelques techniques de programmation

3.5.1. Opérateurs dépendant de paramètres. Indexation. On est souvent amené à considérer un tableau de valeurs d'une grandeur dépendant d'un ou de plusieurs paramètres. Si l'on programme par la méthode opératorielle, on inclut dans l'algorithme de résolution les opérateurs dépendant de paramètres et les opérateurs de variation. Cela fait apparaître, dans les programmes, des instructions à adresses dépendant de paramètres.

Dans les calculateurs modernes on prévoit pour la modification d'adresses des éléments spéciaux appelés registres d'index. Dans certaines instructions quelques positions du code sont réservées à l'indexation (i.e. au numéro d'un des registres d'index). Les adresses réelles s'obtiennent en ajoutant aux adresses relatives de l'instruction en question le contenu du registre d'index correspondant. Le registre d'instruction contient dès maintenant l'instruction modifiée.

Cette technique de programmation, qui utilise la possibilité d'obtenir une adresse réelle en additionnant dans l'unité de commande une adresse relative et le contenu d'un registre d'index, s'appelle *indexation*.

Au cours de l'exécution d'un programme, les instructions dépendant de paramètres restent intactes dans la mémoire principale. La modification s'opère après transfert dans le registre d'instruction.

Exemple 3.4. Considérons le calculateur C-II muni de registres d'index numérotés par $\rho_0, \rho_1, \rho_2, \ldots$ Convenons que $\rho_0 = 0$ et que le contenu de ce registre est toujours nul. Supposons que l'unité de commande du calculateur possède un registre d'instruction r où l'on distingue une zone σ pour le code d'opération θ, une zone i pour l'adresse de l'opérande z et une zone j pour le numéro du registre d'index ρ. Soit $\theta z \rho$ la forme de l'instruction. De plus, supposons que l'unité de commande contient un compteur ordinal l, un registre logique à un chiffre ω et un registre de débordement à un chiffre φ. Désignons les cases de la mémoire principale par $z_1, z_2, \ldots, z_n$. Supposons enfin que l'unité de calcul comporte un registre spécial s.

Un fragment du code d'instruction de C-II est donné dans la table 3.2.

En plus d'instructions utilisées dans les programmes, C-II doit permettre certaines opérations à partir du pupitre de commande, par exemple la mise en marche initiale (réalisée par pression du

Table 3.2

Fragment du code d'instructions de C-II

Code d'opération	Instruction	Description des actions de C-II
01	Addition	$l:=l+1$; $i:=i+\rho_j$; $s:=s+z_i$; $\omega:=s<0$; $\varphi:=\varepsilon(s, 01)$; $l:=(1-\varphi)\cdot l+\varphi\cdot 7777$; $r:=z_i$;
02	Soustraction	$l:=l+1$; $i:=i+\rho_j$; $s:=s-z_i$; $\omega:=s<0$; $\varphi:=\varepsilon(s, 02)$; $l:=(1-\varphi)\cdot l+\varphi\cdot 7777$; $r:=z_i$;
03	Multiplication	$l:=l+1$; $i:=i+\rho_j$; $s:=s\cdot z_i$; $\omega:=s<0$; $\varphi:=\varepsilon(s, 03)$; $l:=(1-\varphi)\cdot l+\varphi\cdot 7777$; $r:=z_i$;
04	Division	$l:=l+1$; $i:=i+\rho_j$; $s:=s:z_i$; $\omega:=s<0$; $\varphi:=\varepsilon(s, 04)$; $l:=(1-\varphi)\cdot l+\varphi\cdot 7777$; $r:=z_i$;
05	Soustraction des valeurs absolues	$l:=l+1$; $i:=i+\rho_j$; $s:=\|s\|-\|z_i\|$; $\omega:=s<0$; $\varphi:=0$; $r:=z_i$;
06	Transfert dans l'additionneur	$l:=l+1$; $i:=i+\rho_j$; $s:=z_i$; $\omega:=s<0$; $\varphi:=0$; $r:=z_i$;
07	Transfert dans une case de mémoire	$l:=l+1$; $i:=i+\rho_j$; $z_i:=s$; $\omega:=s<0$; $\varphi:=0$; $r:=z_i$
10	Transfert dans le registre d'index à partir d'une case de mémoire	$l:=l+1$; $\rho_j:=z_i$; $\varphi:=0$; $r:=z_i$;
11	Transfert de la zone ⌉ d'une instruction dans le registre d'index	$l:=l+1$; $\rho_j:=i$; $\varphi:=0$; $r:=z_i$;

Suite de la table 3.2

Code d'opération	Instruction	Description des actions de C-II
12	Extraction de la racine carrée	$l:=l+1;\ i:=i+\rho_j;$ $s:=\sqrt{\lvert z_i \rvert};\ \varphi:=\varepsilon(s,\ 12);$ $l:=(1-\varphi)\cdot l+\varphi\cdot 7777;$ $r:=z_l;$
13	Branchement inconditionnel	$i:=i+\rho_j;\ l:=i;$ $\varphi:=0;\ r:=z_l;$
14	Branchement conditionnel	$l:=l+1;\ i:=i+\rho_j;$ $l:=\omega\cdot i+(1-\omega)\cdot l;$ $\varphi:=0;\ r:=z_l;$
15	Comparaison du contenu du registre d'index avec la zone adresse de l'instruction	$l:=l+1;\ \omega:=i+\rho_j;$ $\varphi:=0;\ r:=z_l;$
16	Modification du registre d'index	$l:=l+1;\ \rho_j:=\rho_j+i;$ $\varphi:=0;\ r:=z_l;$

bouton-poussoir « mise en marche initiale ») qui fait:

$$\varphi:=0;\quad \omega:=0;\quad l:=0001;\quad r:=z_l;$$

De plus, on a besoin d'une instruction de mise en marche qui consiste à positionner le jeu de tumblers α du pupitre de commande de façon à composer l'adresse de début du programme: puis on presse le bouton-poussoir « mise en marche » et la machine fait:

$$\varphi:=0;\quad \omega:=0;\quad l:=\alpha;\quad r:=z_l;$$

c'est-à-dire commence à exécuter le programme à partir de l'instruction enregistrée dans la cellule z_α.

L'arrêt de C-II est réalisé par programme, sous forme d'une boucle (voir p. 3.5.3) sans issue. L'instruction « mise en marche » fait la machine quitter le régime d'arrêt. On peut prévoir une possibilité d'arrêter le programme par pression du bouton-poussoir « arrêt », du pupitre de commande mais nous n'allons pas considérer cette possibilité.

Tout comme pour C-I, le symbole $\varepsilon(x, y, z, \theta)$ désignera le prédicat « le résultat de l'opération mathématique θ sur x, y s'écarte de celui de l'opération de machine θ ». Pour abréger, nous écrirons

partout $\varepsilon(z, \theta)$ à la place de $\varepsilon(x, y, z, \theta)$, où z est la case où se trouve le résultat de l'opération de machine θ.

EXEMPLE 3.5. Revenons au problème de l'exemple 3.2 où l'on demandait de dresser une table des valeurs de la fonction

$$f(x) = \begin{cases} \varphi(x) & \text{pour } x < x^*, \\ \varphi(x) + \theta(x) & \text{pour } x \geqslant x^* \end{cases}$$

pour $x = x_1, x_2, \ldots, x_n$.

Transformons l'algorithme (3.1) compte tenu des particularités de C-II, il vient :

$$I_0 V_1 \underset{4}{\lrcorner} D_4^i P_3^i \overset{1}{\ulcorner} D_5^i \underset{1}{\lrcorner} P_2 \overset{3}{\ulcorner} V_2 \underset{4}{\llcorner} \underset{3}{\lrcorner} T \, [V_1 i := 1; \; D_4^i y_i := x^5 + x + 1;$$

$$P_3^i x_i < x^*; \; D_5^i y_i := \sqrt{x_i} + 2 + y_i; \; P_2 i = n; \; V_2 i := i + 1;] \quad (3.3)$$

Enfin, les transformations internes des opérateurs (que l'on effectue d'ordinaire mentalement, en même temps que le codage de l'algorithme par les instructions) donnent :

$$I_0 V_1 \underset{4}{\lrcorner} D_6^i P_3^i \overset{1}{\ulcorner} D_7^i \underset{1}{\lrcorner} P_2 \overset{3}{\ulcorner} V_2 \underset{4}{\llcorner} \underset{3}{\lrcorner} T \, [V_1 i := 1; \; D_6^i s := x_i; \; s := s \cdot x_i;$$

$$s := s \cdot x_i; \; s := s \cdot x_i; \; s := s \cdot x_i; \; s := s + x_i;$$

$$s := s + 1; \; y_i := s; \; P_3^i x_i < x^*; \; D_7^i s := \sqrt{x_i};$$

$$s := s + 2; \; s := s + y_i; \; y_i := s; \; P_2 i = n; \; V_2 i := i + 1;] \quad (3.4)$$

Composons un programme pour C-II en nous servant de la méthode d'adresses symboliques. Réservons le régistre d'index 1 au paramètre i, et rangeons les données initiales, comme dans l'exemple 3.2, dans les cases

$$b + 1) \; x_1; \quad b + 2) \; x_2; \; \ldots; \; b + n) \; x_n.$$

Les résultats seront rangés dans les cases

$$d + 1) \; y_1; \quad d + 2) \; y_2; \quad \ldots; \quad d + n) \; y_n$$

et les constantes dans les cases

$$e + 1) \; 1; \quad e + 2) \; 2; \quad e + 3) \; x^*.$$

Les instructions seront disposées dans les cases

$$a + 1, \quad a + 2, \; \ldots$$

Une traduction de l'algorithme dans le langage machine nous fournit le programme suivant en adresses symboliques:

$a+1)$	11	0001	1	V_1
$a+2)$	06	b	1	
$a+3)$	03	b	1	
$a+4)$	03	b	1	
$a+5)$	03	b	1	$\underset{4}{\lrcorner} D_6^1$
$a+6)$	03	b	1	
$a+7)$	01	b	1	
$a+10)$	01	$e+1$	0	
$a+11)$	07	d	1	
$a+12)$	06	b	1	P_3^1
$a+13)$	02	$e+3$	0	
$a+14)$	14	$k+1$	0	$\overset{1}{\ulcorner}$
$a+15)$	12	b	1	
$a+16)$	01	$e+2$	0	D_7^1
$a+17)$	01	d	1	
$a+20)$	07	d	1	
$k+1)$	15	n	1	$\underset{1}{\lrcorner} P_2$
$k+2)$	14	$l+1$	0	$\overset{3}{\ulcorner}$
$k+3)$	16	0001	1	V_2
$k+4)$	13	$a+2$	0	$\underset{4}{\llcorner}$
$l+1)$	13	$l+1$	0	arrêt.

En composant l'instruction $a+14$ nous avons introduit une nouvelle lettre k, parce que nous n'avons pas eu à l'avance le numéro de la case initiale de l'opérateur P_2. Il en est de même de la lettre l dans l'instruction $k+2$. En répartissant les mémoires il faut poser

$$k = a + 20,$$
$$l = k + 4.$$

Les autres lettres peuvent être choisies de manière quelconque.

3.5.2. Programmation en adresses relatives. Bases. Il est parfois commode d'avoir un programme qu'il soit possible de manipuler comme un tout sans avoir à changer la disposition relative des infor-

mations contenues dans le bloc. Ce bloc peut être inclus en tant que sous-programme dans différents programmes de traitement. Il s'agit de programmation en adresses relatives.

Une adresse relative est une adresse comptée à partir d'une origine dont l'adresse absolue est différente de 0. On peut attribuer les adresses relatives non seulement aux instructions d'un programme, mais aussi à d'autres éléments (données initiales, résultats définitifs ou intermédiaires), etc.

La programmation en adresses relatives impose une transformation du programme lors de sa mise en mémoire du calculateur.

Cette opération devient particulièrement simple dans le cas où la machine est munie de registres d'index. Pour placer un programme dans la mémoire à partir d'une case α, il suffit que les adresses relatives soient indexées par le numéro d'un registre β par exemple et que ce registre β contienne le nombre $\alpha - 1$; c'est-à-dire que pour incorporer notre programme dans un autre, il faut le faire précéder de l'instruction envoyant le nombre $\alpha - 1$ dans le registre β.

Cependant il est impossible de modifier par simple indexation les adresses relatives dépendant de paramètres. On peut y remédier si le calculateur permet une double indexation d'adresses des instructions. On utilise à cette fin deux registres, l'un d'*index* et l'autre de *base*. Le contenu du registre de base s'appelle *base*. L'adresse relative inscrite dans l'instruction est appelée *déplacement*. Lors de l'exécution de l'instruction l'adresse réelle s'obtient en additionnant la base, le déplacement et l'index. La somme du déplacement et de l'index est justement l'adresse relative.

EXEMPLE 3.6. Supposons que l'unité de commande d'un calculateur que nous appelons C-III contienne un compteur ordinal l, un registre d'instruction r réunissant les cellules σ, i, j, d, destinées à garder la fonction θ, le numéro du registre d'index, celui du registre base et le déplacement (l'instruction a alors la forme $\theta\rho\rho' b$), un registre k d'adresses réelles, un registre logique ω à un chiffre, un registre de débordement de capacité φ à un chiffre. Munissons en outre l'unité de calcul de C-III d'un registre s. Attribuons au symbole ε (s, θ) la même signification que dans l'exemple 3.4. Un fragment du code d'instructions du C-III est donné dans la table 3.3.

On convient toujours que l'adresse nulle du registre d'index ou du registre de base signifie qu'il n'y a pas d'indexation.

En plus des instructions utilisées dans les programmes, C-III admet certains ordres envoyés à partir du pupitre de commande, par exemple : a) la mise en marche initiale (effectuée par pression du bouton-poussoir « mise en marche initiale ») qui fait $\varphi := 0$; $\omega := 0$; $l := 0001$; $r := z_l$ et b) la mise en marche (on compose l'adresse l à l'aide du jeu de tumblers α et on presse le bouton-pous-

Table 3.3

Fragment du code d'instructions de C-III

Code d'opération	Instruction	Opérations exécutées par la machine
01	Addition	$l := l + 1$; $k := \rho_i + \rho_j + d$; $s := s + z_k$; $\omega := s < 0$; $\varphi := \varepsilon(s, 01)$; $l := (1 - \varphi) \cdot l + \varphi \cdot 7777$; $r := z_l$;
02	Soustraction	Idem (sauf que $s := s - z_k$, $\varphi := \varepsilon(s, 02)$)
03	Multiplication	Idem (sauf que $s := s \cdot z_k$, $\varphi := \varepsilon(s, 03)$)
04	Division	Idem (sauf que $s := s : z_k$, $\varphi := \varepsilon(s, 04)$)
05	Extraction de la racine carrée	$l := l + 1$; $k := \rho_i + \rho_j + d$; $s := \sqrt{z_k}$; $\omega := \|s\| \geqslant 1$; $\varphi := \varepsilon(s, 05)$; $l := (1 - \varphi) \cdot l + \varphi \cdot 7777$; $r_l := z_l$;
06	Transfert d'une case de mémoire dans l'additionneur	$l := l + 1$; $k := \rho_i + \rho_j + d$; $s := z_k$; $\omega := s < 0$; $\varphi := 0$; $r := z_l$;
07	Transfert de l'additionneur dans une case de mémoire	$l := l + 1$; $k := \rho_i + \rho_j + d$; $z_k := s$; $\omega := s < 0$; $\varphi := 0$; $r := z_l$;
10	Envoi du contenu de l'additionneur dans un registre d'index	$l := l + 1$; $\rho_i := s$; $\omega := s < 0$; $\varphi := 0$; $r := z_l$;
11	Envoi du contenu d'un registre d'index dans l'additionneur	$l := l + 1$; $s := \rho_i$; $\omega := s < 0$; $\varphi := 0$; $r := z_l$;
12	Envoi du contenu du registre d'instruction dans l'additionneur	$l := l + 1$; $s := d$; $\omega := s < 0$; $\varphi := 0$; $r := z_l$;

Suite

Code d'opération	Instruction	Opérations exécutées par la machine
13	Addition des paramètres	$l:=l+1$; $\rho_l:=\rho_l+\rho_j$; $\omega:=\rho_l<0$; $\varphi:=0$; $r:=z_l$;
14	Soustraction des paramètres	Idem (sauf que $\rho_l:=\rho_l-\rho_j$)
15	Multiplication des paramètres	Idem (sauf que $\rho_l:=\rho_l\cdot\rho_j$)
17	Branchement conditionnel	$l:=l+1$; $k:=\rho_l+\rho_j+d$; $l:=(1-\omega)\cdot l+\omega\cdot k$: $\omega:=0$; $\varphi:=0$; $r:=z_l$;
20	Instruction de comparaison des contenus des registres d'index et de base	$l:=l+1$; $\omega:=\rho_l+\rho_j$; $\varphi:=0$; $r:=z_l$;
21	Branchement inconditionnel	$k:=\rho_l+\rho_j+d$; $l:=k$; $\varphi:=0$; $r:=z_l$;

soir « mise en marche »); on a

$$\varphi:=0; \quad \omega:=0; \quad l:=\alpha; \quad r:=z_l;$$

EXEMPLE 3.7. Soit à rédiger un programme produisant une table des valeurs de la fonction $y=f(x)$ pour $x=x_1, x_2, \ldots, x_n$ par la méthode d'indexation; ici comme dans les exemples 3.2 et 3.5

$$f(x)=\begin{cases} x^5+x+1 & \text{pour } x<x^*, \\ x^5+x+1+\sqrt{x}+2 & \text{pour } x\geqslant x^*. \end{cases}$$

Nous pouvons nous servir de l'algorithme (3.4) sans modifications. Supposons comme avant que les instructions sont disposées dans les cases $a+1$, $a+2$, . . ., les données initiales dans les cases $b+1$, $b+2$, . . ., les résultats définitifs dans les cases $d+1$, $d+2$, . . . Les constantes seront rangées comme suit: $e+1$) 1; $e+2$) 2; $e+3$) x^*.

Supposons encore que les fonctions des sept registres de manœvre sont distribuées de la façon suivante:

1) a; 2) b; 3) e; 4) d; 5) n; 6) 1; 7) 1.

Ici le contenu du registre n° 6 représente la valeur (pour le moment, initiale) du paramètre *i*, et celui du registre n° 7, une constante servant à modifier la valeur du paramètre *i*. Comme la valeur initiale de *i* est donnée, nous omettons l'opérateur V_1.

Le programme en adresses relatives aura la forme :

1)	06	2	6	0	
2)	03	2	6	0	
3)	03	2	6	0	
4)	03	2	6	0	$\underset{4}{\lrcorner}\ D_6^1$
5)	03	2	6	0	
6)	01	2	6	0	
7)	01	3	0	1	
10)	07	4	6	0	
11)	06	2	6	0	P_3^1
12)	02	3	0	3	
13)	17	1	0	20	$\overset{1}{\ulcorner}$
14)	05	1	6	0	
15)	01	3	0	2	D_7^1
16)	01	2	6	0	
17)	07	4	6	0	
20)	20	5	6	0	$\underset{1}{\lrcorner}\ P_2$
21)	17	1	0	24	$\overset{3}{\ulcorner}$
22)	13	6	7	0	V_2
23)	21	1	0	1	$\underset{4}{\llcorner}$
24)	21	1	0	24	$\underset{3}{\lrcorner}$ arrêt.

Pour ranger le programme obtenu dans la mémoire principale, il faut choisir les valeurs des lettres *a*, *b*, *d*, *e* et faire précéder le programme des instructions envoyant ces valeurs dans les registres correspondants. De plus, il faut placer dans les registres 5, 6 et 7 les nombres *n*, 1 et 1 respectivement.

3.5.3. Organisation de boucles. L'organisation de boucles est un des procédés les plus importants de programmation. Il permet d'avoir des programmes qui assurent l'exécution d'une quantité d'instructions dépassant de beaucoup le nombre des instructions dans les programmes mêmes. Cela est possible grâce à ce que cer-

taines séquences d'instructions du programme s'exécutent de nombreuses fois, avec éventuellement des modifications nécessaires. De telles séquences d'instructions s'appellent *boucles* de programme.

Puisque dans le présent chapitre nous sommes guidés par la méthode de programmation opératorielle où les éléments d'un programme sont considérés comme des images des opérateurs du langage des schémas logiques, nous préférons étudier les boucles non pas dans les programmes, mais dans les algorithmes en YALS. Précisons donc la notion de boucle.

On appelle *boucle structurale sans branchement* une expression, faisant partie d'un algorithme, qui :

1) contient un signe de passage fermant à gauche du signe de passage ouvrant de même indice ;

2) du point de vue syntaxique, admet l'exécution de certains opérateurs après les opérateurs qui les suivent (au sens de l'ordre d'exécution) ou bien contient un seul opérateur ;

3) est telle que si l'on rejette la fermeture de n'importe quel de ses opérateurs la condition 1) ou 2) n'est plus remplie.

Disons en passant que la troisième condition est celle de minimalité d'une boucle structurale sans branchement, selon laquelle il est impossible de compléter une telle boucle par d'autres opérateurs.

EXEMPLE 3.8. L'expression

$$D_1 \underset{1}{\lrcorner} D_2P_2 \underset{3}{\llcorner} D_3 \tag{3.5}$$

n'est pas une boucle structurale sans branchement car la condition 1) n'est pas satisfaite.

Pour les expressions

$$D_1 \underset{1}{\lrcorner} D_2 \underset{1}{\llcorner} P_3 \tag{3.6}$$

$$\underset{1}{\lrcorner} D_2 \underset{2}{\llcorner} \underset{3}{\lrcorner} D_1 \underset{4}{\llcorner} \underset{2}{\lrcorner} P_1 \underset{1}{\llcorner} \tag{3.7}$$

c'est la condition 3) qui n'est pas satisfaite (on peut rejeter les fermetures des opérateurs D_1 et D_3). Il en est de même de l'expression

$$\underset{1}{\lrcorner} D_1P_1 \underset{2}{\llcorner} D_2 \underset{2}{\lrcorner} P_2 \underset{1}{\llcorner} \tag{3.8}$$

où, sans nuire à la condition 1), on peut rejeter la fermeture de l'opérateur D_2.

Mais l'expression

$$\underset{1}{\lrcorner} D_1P_1 \underset{1}{\llcorner} \tag{3.9}$$

est bien une boucle structurale sans branchement.

Disons en anticipant que (3.5) n'est pas du tout une boucle, que (3.6) et (3.7) contiennent, en plus d'une boucle structurale

sans branchement, d'autres opérateurs et que la boucle (3.8) est une boucle branchée.

On appelle *boucle structurale* une expression opératorielle représentant une boucle structurale sans branchement ou une expression qui se réduit à une telle boucle si l'on réunit certains de ses opérateurs en un opérateur généralisé et que l'on considère ce dernier au même titre que les autres opérateurs.

Exemple 3.9. L'expression (3.8) de l'exemple précédent est une boucle structurale. On peut la mettre sous la forme

$$\underset{1}{\lrcorner} R (\underset{2}{\llcorner}) \underset{2}{\lrcorner} P_2 \underset{1}{\llcorner} \qquad (3.10)$$

qui satisfait à la définition de la boucle structurale sans branchement si l'on considère R comme un opérateur comme les autres. Ici

$$R (\underset{2}{\llcorner}) = D_1 P_1 \underset{2}{\llcorner} D_2$$

L'expression

$$\underset{1}{\lrcorner} D_1 \underset{2}{\lrcorner} D_2 P_1 \underset{2}{\llcorner} P_2 \underset{1}{\llcorner} \qquad (3.11)$$

est aussi une boucle structurale, puisque, en posant

$$\underset{2}{\lrcorner} D_2 P_1 \underset{2}{\llcorner} = R$$

nous réduisons (3.11) à la forme

$$\underset{1}{\lrcorner} D_1 R P_2 \underset{1}{\llcorner}$$

Notons que l'expression R elle-même représente une boucle structurale sans branchement, de sorte que la boucle structurale (3.11) contient en elle une autre boucle structurale R.

On appelle *boucle* un ensemble de fermetures d'opérateurs qui peut être obtenu à partir d'une boucle structurale moyennant une série de transformations équivalentes déplaçant les fermetures d'opérateurs (voir § 4.7, transformation (4.16)).

Il découle de cette définition que chaque boucle structurale est une boucle car laisser une fermeture d'opérateur en place représente un cas particulier d'une transformation de déplacement.

On ne considère dans le présent chapitre que les boucles structurales, le système des transformations équivalentes n'étant étudié que dans le chapitre suivant.

Dans toute boucle, on trouve un ou plusieurs opérateurs logiques qui commandent l'exécution des opérations de la boucle. On peut reconnaître un opérateur logique de commande par le fait qu'il contient un signe de passage ouvrant extérieur.

Examinons quelques exemples simples de boucles contenant un seul opérateur logique de commande.

3.5.3.1. Boucle itérative élémentaire. Une boucle itérative élémentaire contient deux opérateurs d'action et un opérateur logique. Elle est de la forme

$$\pi \underset{m}{\lrcorner} D_1 D_2 P_1 \lambda_m \bar{\lambda}_j,$$

où π est un ensemble de signes de passage fermants, $\bar{\lambda}_j$ désigne un signe de passage ouvrant supérieur (inférieur) si λ_m est un signe de passage ouvrant inférieur (supérieur). De plus, le premier des opérateurs d'action est un opérateur de transfert d'information d'un groupe de cellules dans un autre, le second opérateur envoie ses résultats dans le premier groupe de cellules. L'opérateur logique vérifie une condition qui dépend du contenu de deux groupes de cellules.

Exemple 3.10. L'équation

$$x = f(x)$$

peut être résolue par une méthode itérative si l'on connaît une valeur approchée x_0 de la racine cherchée. La méthode consiste à calculer les nombres

$$x_i = f(x_{i-1})$$

pour $i = 1, 2, \ldots$ Pour chaque nouvelle valeur x_i de la racine on vérifiera la condition

$$|x_i - x_{i-1}| < \varepsilon,$$

où ε est un petit nombre positif. On arrête les calculs dès que la condition se trouve satisfaite.

Pour résoudre le problème on peut adopter l'algorithme

$$I_0 D_3 \underset{1}{\lrcorner} D_1 D_2 P_1 \underset{1}{\llcorner} \quad T_1 [D_1 \beta := \alpha; D_2 \alpha := f(\beta); D_3 \alpha := x_0; P_1 |\alpha - \beta| < \varepsilon;]$$

qui contient une boucle itérative.

3.5.3.2. Boucle commandée par une variable. Lorsqu'une boucle est commandée par un seul opérateur logique, et que les opérateurs de la boucle modifient un seul argument du prédicat vérifié par l'opérateur logique en question, alors l'argument mentionné s'appelle *variable de commande*, ou *variable de contrôle*. Dans l'exemple 3.10, l'opérateur commandant les exécutions de la boucle vérifiait le prédicat dont les deux arguments étaient modifiés par les opérateurs de la boucle.

Un cas spécial de boucle commandée par une variable est celui où la variable de contrôle est un paramètre. Son schéma peut avoir la forme

$$\underset{m}{\lrcorner}\, V_i D_1^i P_i \lambda_m \bar{\lambda}_j \tag{3.12}$$

(la signification des symboles utilisés est expliquée au p. 3.5.3.1).

L'opérateur V_i modifie la valeur du paramètre i dont dépend l'opérateur D_1^i, et l'opérateur P_i vérifie un prédicat $\theta(i)$. Souvent V_i désigne $i := i+1$; et l'opérateur P_i représente le prédicat $i < n$ ($n = \text{const}$). Alors (3.12) a la forme

$$\underset{m}{\lrcorner}\, i := i+1;\ D_1^i i < n;\ \lambda_m \bar{\lambda}_j \tag{3.13}$$

Pour entrer dans la boucle, une initialisation est nécessaire. En dehors de la boucle on prévoit un opérateur $i := c$; qui détermine la valeur initiale du paramètre i. Si $c = 0$, la boucle (3.13) s'exécute n fois.

Si dans un programme, l'opérateur D_1^i exprime une dépendance de i des adresses d'une instruction, la boucle (3.13) s'appelle boucle avec modification d'adresses. Une telle boucle peut avoir un autre schéma :

$$\underset{m}{\lrcorner}\, D_1^i i := i+1\ ;\ i < n\ ;\ \lambda_m \bar{\lambda}_j. \tag{3.14}$$

Pour que la boucle (3.14) réalise les mêmes calculs que (3.13), l'opérateur d'initialisation du paramètre doit avoir la forme $i := c+1$;

Dans le cas

$$i := c;\ \underset{m}{\lrcorner}\, i := i+1\ ;\ D_1 i < n\ ;\ \lambda_m \bar{\lambda}_{.} \tag{3.15}$$

ou bien

$$\underset{m}{\lrcorner}\, i := i+1\ ;\ D_1^i i < n\ ;\ \lambda_m i := c;\ \ \underset{\vdots}{\llcorner} \tag{3.16}$$

l'opérateur $i := c$; s'appelle *opérateur de restitution* du paramètre i, et les expressions (3.15) et (3.16) s'appellent boucles avec restitution du paramètre de commande. Il est clair que des schémas analogues existent pour la boucle (3.14).

Dans la boucle schématisée dans (3.13) ou (3.14), commandée par le paramètre i, les opérateurs $i := i+1$; et $i < n$; forment ce qu'on appelle *compteur d'exécutions de boucle*.

Exemple 3.11. L'algorithme (3.1) de calcul des valeurs de $f(x_i)$, $i = 1, 2, \ldots, n$ comprend la boucle

$$\underset{\cdot}{\lrcorner}\, D_1^i P_1^i \underset{1}{\llcorner}\, D_2^i \underset{2}{\llcorner}\ \underset{1}{\lrcorner}\, D_3^i \underset{2}{\lrcorner}\, P_2 \overset{3}{\ulcorner}\, V_2 \underset{4}{\llcorner}$$

commandée par l'opérateur P_2 vérifiant la condition $i = n$; dans cette boucle, l'opérateur logique de commande n'est pas le dernier. On pourrait écrire la boucle sous la forme

$$\underset{4}{\rightharpoondown} D_1^i P_1^i \underset{1}{\llcorner} D_2^i \underset{2}{\llcorner} \underset{1}{\lrcorner} D_3^{ii} \underset{2}{\lrcorner} V_2 P_2^i \overset{3}{\underset{4}{\sqsubset}}$$

Ici l'opérateur P_2^i vérifie le prédicat $i - 1 = n$; ou bien, ce qui revient au même, $i = n + 1$; L'opérateur P_1^i ne commande pas la boucle, mais réalise un branchement. Remarquons que le passage de la première expression à la deuxième implique une modification du programme.

Notons qu'actuellement, pour construire des boucles commandées par paramètres dans les programmes pour les calculateurs modernes munis de registres d'index on emploie couramment le procédé d'indexation décrit plus haut.

3.5.4. Echelles logiques. Soit un opérateur Q à qui on passe la commande un nombre donné de fois (par exemple, n) au cours de l'exécution du programme. Supposons encore qu'après la i-ème exécution de l'opérateur Q, la commande doit passer à un opérateur de numéro N_i. Autrement dit, on donne une fonction à valeurs numériques entières

$$N_i = f(i).$$

Ecrivons toutes les valeurs de cette fonction $N_1, N_2, \ldots, N_n$ et formons-en une suite de nombres deux à deux différents $M_1, M_2, \ldots$ $\ldots, M_{n'}$ (telle que pour tout N_i il existe un M_j qui lui soit égal, et pour tout M_j il existe au moins un N_i égal).

Définissons le nombre entier k de la manière suivante:

$$2^{k-1} \leqslant n' \leqslant 2^k - 1,$$

c'est-à-dire

$$\log_2 (n' + 1) \leqslant k \leqslant 1 + \log_2 n'.$$

Considérons tous les entiers binaires à k chiffres et choisissons-en arbitrairement n' nombres. A tout nombre M_j faisons correspondre, d'une manière biunivoque, l'un des nombres choisis m_j. Puis faisons correspondre à chaque N_i le nombre binaire m_{j_i} qui correspond à M_j égal à N_i.

La suite de chiffres binaires que l'on obtient en écrivant successivement les nombres binaires $m_{j_1}, m_{j_2}, \ldots, m_{j_n}$ s'appelle *échelle logique.*

En séparant les k premiers chiffres de l'échelle logique après la première exécution de l'opérateur Q et en les comparant successivement avec les nombres $m_1, m_2, \ldots, m_{n'}$, on peut reconnaître le numéro de l'opérateur auquel la commande doit passer. Il faut en-

suite éliminer les k premiers chiffres de l'échelle et écrire à droite un nombre binaire à k chiffres, autre que m_j. Cela se fait pour garder constant le nombre total de chiffres de l'échelle et pour pouvoir saisir le moment où l'échelle est épuisée. Après ces opérations l'échelle est prête à une nouvelle exécution de l'opérateur Q.

D'habitude, on établit la correspondance biunivoque entre les nombres binaires m_j et les nombres M_j ($j = 1, 2, \ldots, n'$) d'une telle manière qu'au numéro de l'opérateur N_n il corresponde un nombre binaire non nul. Alors, en effaçant les chiffres utilisés de l'échelle, on peut la compléter à droite par le zéro à k chiffres. L'échelle est épuisée dès que tous ses chiffres deviennent nuls. On voit que la transformation que subit l'échelle après la séparation des k premiers chiffres est équivalente dans ce cas au décalage de k positions à gauche.

Souvent il est préférable d'effectuer d'abord un décalage de l'échelle et puis la séparation des k premiers chiffres. Pour que cela soit possible, on écrit immédiatement avant le nombre m_{j_1} de l'échelle un nombre quelconque m_{j_0} à k chiffres.

La réalisation technique d'une échelle logique dépend des valeurs des nombres k et n. En géneral, on range l'échelle dans des cases successives de la mémoire, et pour séparer ses k premiers chiffres (au moyen de l'intersection logique), on utilise une suite de chiffres binaires dont les k premières positions sont des unités, les autres des zéros. Cette suite s'appelle indicateur d'échelle. Dans le cas le plus simple, l'échelle ainsi que son indicateur occupe une case de mémoire.

Une comparaison des nombres binaires à k chiffres, séparés dans l'échelle, avec les nombres $m_1, m_2, \ldots, m_{n'}$ et les passages de commande sont réalisés par un groupe d'opérateurs logiques qu'on dispose immédiatement après l'opérateur Q.

EXEMPLE 3.12. Soit un] schéma logique

$$I_0 D_1 D_2 D_3^{j} V_4 \ldots D_7 D_8 T_1,$$

où les points de suspension sont mis au lieu des opérateur logiques nécessaires à l'utilisation de l'échelle logique. Les passages de commande après la i-ème exécution de l'opérateur V_4 sont indiqués dans la table 3.4.

Table 3.4

Valeurs de la fonction à valeurs entières N_i

i	1 à 6	7	8 ; 9	10
N_i	3	4	3	7

Il faut construire l'échelle logique assurant les passages de commande indiqués et inclure dans le schéma les opérateurs logiques nécessaires.

On voit dans la table 3.4 que $M_1 = 3$, $M_2 = 4$, $M_3 = 7$, donc $n' = 3$. On a $k = 2$ (puisque $2^1 \leqslant 3 \leqslant 2^2 - 1$). Il existe en tout quatre nombres binaires à deux chiffres. Choisissons-en trois (parce que $n' = 3$) :

$$00,\ 01,\ 10.$$

Posons $m_1 = 00$, $m_2 = 01$, $m_3 = 10$, $m_4 = 11$ (m_0 arbitraire). L'échelle logique aura la forme

$$11\ 00\ 00\ 00\ 00\ 00\ 00\ 01\ 00\ 00\ 10.$$

Si l'échelle est à utiliser pour un calculateur dont le mot machine est à 43 positions, il faut ajouter à droite des zéros au nombre de 21. On aura

$$11\ 00\ 00\ 00\ 00\ 00\ 00\ 01\ 00\ 00\ 10\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 0.$$

L'indicateur de l'échelle sera alors de la forme

$$1100000000000000000000000000000000000000000.$$

Si l'on range l'échelle dans la case $c + 1$, l'indicateur dans la case $c + 2$, et que l'on réserve la case $c + 3$ pour le code

$$01\ 00000000000000000000000000000000000000000,$$

alors pour l'utilisation de l'échelle on aura évidemment besoin d'un opérateur d'action D_9 et de deux opérateurs logiques P_5 et P_6. L'opérateur D_9 doit décaler l'échelle de deux positions à la gauche et séparer, au moyen de l'indicateur, ses deux premiers chiffres. Si l'on obtient ainsi une collection des zéros, l'opérateur P_5 doit passer la commande à l'opérateur D_3^j. Dans le cas contraire la commande passe à l'opérateur P_6 qui compare la collection séparée de chiffres avec le contenu de la case $c + 3$. S'il y a égalité, la commande est à passer à l'opérateur V_4, sinon à l'opérateur D_7. Ainsi le schéma logique (3.15) prend la forme

$$I_0 D_1 D_2 \underset{1\,1}{\lrcorner} D_3^j \underset{2}{\lrcorner} V_4 D_9 P_5 \overset{1}{\ulcorner} P_6 \overset{2}{\ulcorner} D_7 D_8 T_1$$

3.5.5. Sous-programme

3.5.5.1. Notions de sous-schéma et de sous-programme. Constitution d'un sous-programme. Soit

$$\textstyle\sum_1 \doteq \sum \{W_1, W_2, \ldots, W_n\}$$

le schéma logique d'un programme contenant les expressions $W_1, W_2, \ldots, W_n$. Supposons qu'une analyse du schéma ait établi l'équivalence de chacune des expressions $W_2, W_3, \ldots, W_n$ à l'ex-

pression W_1. Remplaçons dans le schéma $\sum_1$ chaque expression W_k ($k \geqslant 2$) par le symbole correspondant E_k qui désigne le transfert de la commande à W_1 et le retour de la commande à W_k correspondant après l'exécution de W_1. Ce symbole est *opérateur d'appel de* W_1. Le schéma $\sum \{W_1, W_2, \ldots, W_n\}$ se transforme alors en un schéma équivalent

$$\sum\nolimits_2 = \sum \{W_1, E_2, E_3, \ldots, E_n\}.$$

Dans ce cas l'expression W_1 s'appelle *sous-schéma*, et la transformation décrite du schéma logique initial s'appelle *formation d'un sous-schéma.*

Le schéma logique $\sum_2$ peut aussi bien être mis sous la forme

$$\sum\nolimits_3 = \sum \{E_1, E_2, E_3, \ldots, E_n\} W_1,$$

où E_1 est aussi un opérateur d'appel de W_1. Cette transformation s'appelle *séparation* du sous-schéma W_1.

Soient S_1, S_2 et S_3 les programmes correspondant aux schémas logiques $\sum_1$, $\sum_2$ et $\sum_3$, alors aux expressions $W_1, W_2, \ldots, W_n$ figurant dans $\sum_1$ correspondent dans le programme S_1 certaines parties $A_1, A_2, \ldots, A_n$.

Dans le programme S_2, les parties correspondant aux expressions $W_2, W_3, \ldots, W_n$ seront remplacées par les groupes d'instructions qui réalisent les appels à la partie A_1 correspondant à W_1. Dans ce cas, A_1 s'appelle *sous-programme.*

Dans la construction du programme S_2 est utilisé un procédé de programmation qu'on appelle *formation de sous-programme.*

En ce qui concerne le programme S_3 correspondant au schéma logique $\sum_3$, on dit qu'on y a effectué une *séparation* du sous-programme.

Dans les cas où les expressions W_i se composent de plusieurs opérateurs chacune, la formation d'un sous-schéma simplifie la notation du schéma logique et le rend plus clair.

3.5.5.2. Appels d'un sous-programme. Soient $A_1, A_2, \ldots, A_n$ les parties du programme S_1 correspondant aux expressions $W_1, W_2, \ldots$ $\ldots, W_n$ dans le schéma $\sum_1$. Les instructions d'une partie A_i qui reçoivent la commande de l'extérieur seront appelées *entrées*, et celles qui rendent la commande à l'extérieur seront appelées *sorties.* Dans le cas où il est possible de séparer le sous-programme A_1, à toute son entrée correspondra une entrée de chaque partie A_i ($i \geqslant 2$).

De telles entrées sont dites *semblables.*

Tout à fait de même, à chaque sortie de A_1 il correspond une certaine sortie de A_i.

On dit que ces sorties sont *semblables.*

Un groupe d'instructions qui réalise un appel du sous-programme et qui correspond à un opérateur d'appel du schéma logique du programme est appelé opérateur d'appel élémentaire.

On peut faire de A_1 un sous-programme de la manière suivante :

1. Réserver une case « libre » à la suite de chaque sortie de A_1 qui n'est pas une instruction de branchement conditionnel ou inconditionnel (une telle sortie passe la commande à la case immédiatement suivante). Ces cases libres sont destinées à y mettre les instructions de retour et on les considère par la suite comme sorties du sous-programme A_1.

2. Remplacer chaque partie A_i du programme par un opérateur d'appel de A_1. Un opérateur d'appel doit prévoir :

— l'envoi des données initiales nécessaires dans certaines cases d'où le sous-programme les « extrait » pour ses calculs (si les données initiales n'y sont pas enregistrées par d'autres opérateurs du programme) ;

— l'envoi aux sorties du sous-programme d'instructions de retour, i.e. d'instructions qui rendent la commande aux instructions qui l'auraient reçue à partir des sorties semblables de A_i ;

— le transfert de la commande à l'entrée correspondante du sous-programme ;

— parfois, à la fin du travail du sous-programme, l'effacement des instructions de retour ou la restauration d'anciennes instructions de retour et éventuellement, le transfert des résultats dans d'autres cases.

Il est parfois utile de conférer toutes ces tâches au groupe d'instructions d'appel du sous-programme. Alors ce groupe n'est plus un opérateur puisqu'il possède plusieurs entrées et sorties. Tout de même, pour des raisons de commodité, on peut le considérer comme un opérateur élémentaire. Le plus facile est l'appel des sous-programmes ayant une seule entrée et un seul signe de passage ouvrant à la sortie, ou bien des sous-programmes dans lesquels à chaque entrée correspond un seul signe de passage ouvrant à la sortie. Il faut faire attention à ce que chaque signe de passage ouvrant à la sortie d'un sous-programme diffère par son indice de tous les autres signes de passage ouvrants. L'opérateur d'appel du sous-programme est désigné, dans le schéma logique qui lui correspond, par le symbole $E_v(\llcorner_\alpha ; \lrcorner_\beta)$, où $\llcorner_\alpha$ est le passage au sous-programme, $\llcorner_\beta$ le retour de la commande au programme principal après la sortie du sous-programme.

Remarquons que l'appel d'un sous-programme n'est pas un opérateur. Pourtant, ensemble avec le sous-programme concerné, il peut être considéré comme opérateur généralisé.

EXEMPLE 3.13. Soit à rédiger un programme de calcul de la quantité

$$y = \sin^2 \left[\sum_{j=1}^{n} \sin x_j \right]$$

d'après les nombres donnés $x_1, x_2, \ldots, x_j, \ldots, x_n$.

On sait que

$$\sin x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Cette série est alternée pour toute valeur de x, donc l'erreur absolue commise en remplaçant la somme de la série par une somme partielle ne dépasse pas la valeur absolue du dernier terme.

Si l'on désigne par u_i le i-ème terme de la série, i.e.

$$u_i = \frac{x^{2i-1}}{(2i-1)!},$$

on a

$$u_{i+1} = u_i \left[-\frac{x^2}{2i(2i+1)} \right]. \qquad (3.17)$$

On peut se servir de la dernière relation pour calculer $\sin x$.

Le schéma logique du programme correspondant aura la forme

$$I_0V_1D_1 \underset{1}{\lrcorner} D_2^jD_3 \underset{2}{\lrcorner} D_4P_1 \underset{2}{\llcorner} D_6V_2P_2 \underset{1}{\llcorner} D_7D_3 \underset{3}{\lrcorner} D_4P_1 \underset{3}{\llcorner} D_8T_1\ [V_1j := 1;$$

$$D_1t := 0;\ D_2^jx := x_j;\ D_3v := x;\ s := x;\ i = 0;\ a := -x^2;\ D_4i := i+1;$$

$$v := v \cdot a : (2i \cdot (2i+1));\ s := s+v;\ P_1 |v| < \varepsilon;\ D_6t := t+s;$$

$$V_2j := j+1;\ P_2j = n+1;\ D_7x := t;\ D_8y := s^2;] \qquad (3.18)$$

En considérant le schéma (3.18) on voit que

$$D_3 \underset{2}{\lrcorner} D_4P_1 \underset{2}{\llcorner} \quad \text{et} \quad D_3 \underset{3}{\lrcorner} D_4P_1 \underset{3}{\llcorner}$$

représentent des parties identiques du programme. En utilisant la première de ces expressions en tant que sous-programme (sous-schéma) nous mettons (3.18) sous la forme

$I_0V_1D_1 \underset{1}{\lrcorner} D_j^2 \underset{4}{\llcorner} \underset{4}{\lrcorner} D_3 \underset{2}{\lrcorner} D_4P_1 \overset{5}{\underset{2}{\sqsubset}} \underset{5}{\lrcorner} D_6V_2P_2 \underset{1}{\llcorner} D_7E_1 (\underset{4}{\llcorner}; \underset{5}{\lrcorner})D_8T_1$ [décodage des opérateurs]

CHAPITRE 4

TRANSFORMATIONS ÉQUIVALENTES DES ALGORITHMES

§ 4.1. Notion d'équivalence des algorithmes. Complexes. Transformations équivalentes

Soient A et B des algorithmes (chacun dans un langage) sur un langage L.

Il est démontré que l'ensemble des résultats que l'on peut obtenir en exécutant un algorithme à partir des données initiales admissibles (constructions du langage L) représente un langage *), qui peut faire partie du langage L ou se confondre avec lui.

Soit

$$L = \{s\},$$

i.e. soit s une construction arbitraire du langage L. Désignons par $\tilde{A}$ et $\tilde{B}$ les applications réalisées respectivement par les algorithmes A et B.

Si, chaque fois que $\tilde{A}(s)$ ou $\tilde{B}(s)$ existent **), a lieu l'égalité

$$\tilde{A}(s) = \tilde{B}(s),$$

on dit que les algorithmes A et B sont *équivalents* et l'on écrit

$$A \sim B.$$

Les langages des résultats de deux algorithmes équivalents se confondent.

Notons qu'il n'est pas d'usage de considérer des algorithmes équivalents comme un même algorithme.

La définition de l'équivalence d'algorithmes que nous venons de formuler, et qui est admise dans la théorie des algorithmes, s'avère trop étroite pour les besoins de la programmation. Nous allons formuler une nouvelle définition de l'équivalence d'algorithmes (chaque fois où ça prête à confusion, nous l'appellerons équivalence généralisée) qui exprimera le fait non formel que les algorithmes résolvent le même problème universel, indépendamment des

*) Cela découle de la définition d'un langage (§ 1.2) et de celle d'une opération (p. 1.5.3.).

**) C'est-à-dire si l'algorithme considéré est applicable à la donnée s.

langages employés, des désignations utilisées et des relations entre les éléments des données initiales (ou des résultats).

Supposons que dans une collection d'opérations $\Omega = \{\omega\}$ il existe deux opérations inverses l'une de l'autre ω_i et ω_j, telles que

$$\omega_i(s) = t, \quad \omega_j(t) = s, \tag{4.1}$$

où t et s appartiennent aux langages L_1 et L_2 respectivement.

Deux constructions s et t satisfaisant à (4.1) sont dites ***équivalentes*** par rapport à Ω. On note

$$s \approx t; \ \Omega. \tag{4.2}$$

Deux algorithmes A et B sur les langages respectifs L_1 et L_2 sont dits ***équivalents*** par rapport à un couple de collections d'opérations Ω, Ω' si, pour les données initiales satisfaisant à (4.2), les résultats

$$\tilde{A}(s) = r, \quad \tilde{B}(t) = \rho \tag{4.3}$$

satisfont aux conditions

$$r \approx \rho; \ \Omega'. \tag{4.4}$$

On écrit alors

$$A = B; \ \Omega, \ \Omega'. \tag{4.5}$$

En d'autres termes, deux algorithmes sont équivalents si l'équivalence de leurs données initiales implique celle des résultats.

Dans le présent chapitre on décrit le cas spécial où les deux algorithmes sont donnés en YALS, et leurs données initiales sont représentées dans un langage des opérandes lié au YALS. De plus, on suppose que les collections Ω et Ω' coïncident et contiennent les opérations simples suivantes:

1) transformation équivalente;

2) changement de noms de cellules sans donner un même nom à des cellules différentes;

3) élimination d'une cellule dont l'état, pour n'importe quelles données initiales, est identique à l'état d'une autre cellule;

4) incorporation d'une cellule dont l'état, pour n'importe quelles données initiales, est identique à celui d'une cellule déjà utilisée;

5) élimination de cellules non essentielles;

6) incorporation de cellules non essentielles;

Dans ces conditions, l'équivalence de deux algorithmes A et B sera notée

$$A = B \tag{4.6}$$

au lieu de la notation (4.5).

En étudiant l'équivalence, on prend les algorithmes avec leurs données initiales et résultats. Un tel ensemble s'appelle ***complexe***.

Toute transformation d'un complexe qui amène à un nouveau complexe dont l'algorithme est équivalent à l'algorithme initial s'appelle *transformation équivalente de l'algorithme* (plus précisément, du complexe).

Remarquons que l'équivalence des algorithmes (complexes) est: 1) *réflexive*, i.e. $A = A$; 2) *symétrique*, i.e. si $A = B$, alors $B = A$; 3) *transitive*, i.e. les conditions $A = B$ et $B = C$ impliquent $A = C$.

§ 4.2. Complexe uniforme. Equivalent d'un complexe uniforme

Un complexe est dit *uniforme* si son algorithme: 1) se termine par l'opérateur T_1, 2) ne contient pas d'autres opérateurs de fin, 3) ne peut contenir que des signes de passage implicites, 4) utilise les cellules d'un seul type (c'est-à-dire soit des objets, soit des paramètres).

Pour abréger les notations, supprimons les opérateurs I_0 et T_1 dans un complexe uniforme. Les opérateurs élémentaires seront écrits en colonne, la tradition veut qu'une telle colonne s'appelle *suite normale de formules*. Les données initiales et les résultats seront représentés par deux ensembles de cellules, X et Y. Nous supposons donné l'ensemble G des états de cellules de X, qu'on appelle domaine de définition du complexe uniforme. Les deux ensembles X et Y, bien que ce ne soit pas essentiel, seront supposés ordonnés d'une façon quelconque. Ces ensembles seront appelés *cortèges*.

Soit

$$\left.\begin{array}{l} \alpha_{t_1} := \varphi_1(\alpha_1, \alpha_2, \ldots, \alpha_n); \\ \alpha_{t_2} := \varphi_2(\alpha_1, \alpha_2, \ldots, \alpha_n); \\ \cdots\cdots\cdots\cdots\cdots \\ \alpha_{t_N} := \varphi_N(\alpha_1, \alpha_2, \ldots, \alpha_n); \end{array}\right\} \qquad (4.7)$$

une suite normale. Nous convenons que ces formules ne contiennent que les cellules $\alpha_1, \alpha_2, \ldots, \alpha_n$. Cela ne veut pas dire du tout que chaque fonction $\varphi_j(\alpha_1, \alpha_2, \ldots, \alpha_n)$ figurant dans (4.7) a effectivement tous les $\alpha_1, \alpha_2, \ldots, \alpha_n$ pour arguments. Il se peut que certains α_i sont des arguments *fictifs*, comme c'est le cas, par exemple, de l'argument y dans la notation $f(x, y)$ désignant $\sin x$. Si un α_i est un argument essentiel d'une fonction φ_j, nous écrirons

$$\alpha_i = sa\varphi_j$$

(se lit: α_i est un argument essentiel de φ_j). Et lorsque α_i est un argument fictif d'une fonction φ_j nous écrirons

$$\alpha_i \neq sa\varphi_j$$

(se lit: α_i n'est pas un argument essentiel de φ_j).

Soient $X = \{\alpha_{s_i}\}_{i=1}^{m}$ le cortège d'entrée et $Y = \{\alpha_{r_i}\}_{i=1}^{M}$ le cortège de sortie *) de la suite (4.7).

Ecrivons tous les arguments essentiels de la fonction φ_1, ensuite, pour tous les $k = 2, 3, \ldots, N$, écrivons tous les arguments essentiels de chaque fonction φ_k qui diffèrent de ceux déjà écrits et de $\alpha_{t_1}, \alpha_{t_2}, \ldots, \alpha_{t_{k-1}}$. Nous obtenons un cortège de cellules que nous désignons par L. De plus, soit L' le cortège formé par toutes les cellules deux à deux différentes qu'on trouve dans la suite α_{t_1}, $\alpha_{t_2}, \ldots, \alpha_{t_N}$. Dans les conditions

$$L \subseteq X, \quad Y \subseteq X + L',$$

si l'état du cortège X est donné, le complexe peut être exécuté ce qui conduit à la détermination de l'état du cortège Y. L'ensemble des cellules appartenant à L' et n'appartenant pas à Y sera appelé *cortège intermédiaire.*

EXEMPLE 4.1. Considérons à titre illustratif un complexe K ayant

$$X = \{x, y, z\}$$

pour cortège d'entrée,

$$\begin{aligned} u &:= x - y; \\ v &:= x + z; \\ v &:= v \cdot u; \\ w &:= y - v; \\ t &:= 1 : w; \\ w &:= t - w; \end{aligned}$$

pour suite normale et

$$Y = \{x, w\}$$

pour cortège de sortie. Son cortège intermédiaire sera

$$Z = \{u, v, t\}$$

On peut considérer comme domaine de définition du complexe l'ensemble des états du cortège X pour lesquels est exécutable l'opération $1 : w$ figurant dans l'avant-dernière formule de la suite normale, i.e. le domaine $y(1 + x + z) \neq x(x + z)$.

Nous admettons les complexes dont les suites normales sont vides. Chaque tel complexe doit satisfaire à la condition $Y \subseteq X$. On pose $N = 0$ (où N est le nombre de formules dans la suite normale).

---

*) Les cellules α_{s_i} et α_{r_i} doivent évidemment se trouver parmi les $\alpha_1, \alpha_2, \ldots, \alpha_n$.

Considérons un complexe K dont le domaine de définition est G, le cortège d'entrée X se compose des cellules $\alpha_{s_1}, \alpha_{s_2}, \ldots, \alpha_{s_m}$, celui de sortie Y se compose des cellules $\alpha_{r_1}, \alpha_{r_2}, \ldots, \alpha_{r_m}$ et la suite normale est

$$\begin{aligned} \alpha_{t_1} &:= \varphi_1(\alpha_1, \alpha_2, \ldots, \alpha_n); \\ \alpha_{t_2} &:= \varphi_2(\alpha_1, \alpha_2, \ldots, \alpha_n); \\ &\cdots\cdots\cdots\cdots \\ \alpha_{t_N} &:= \varphi_N(\alpha_1, \alpha_2, \ldots, \alpha_n); \end{aligned}$$

Réalisons le processus déterminé par l'algorithme de substitutions successivement pour $k = 1, 2, \ldots, M$.

Algorithme de substitutions.

1°. Poser $i = N$. Passer au p. 2°.

2°. Poser $\theta_{k,i} = \alpha_{r_k}$. Passer au p. 3°.

3°. Vérifier si l'égalité $i = 0$ a lieu. Si oui, choisir $\theta_{k,0}$ pour θ_k et terminer le processus; si non, passer au p. 4°.

4°. Substituer la fonction φ_i à α_{t_i} dans $\theta_{k,i}$. Prendre le résultat de la substitution pour $\theta_{k,i-1}$. Passer au p. 5°.

5°. Décrémenter i d'une unité. Passer au p. 3°.

Après avoir exécuté l'algorithme de substitutions pour $k = 1, 2, \ldots, M$, nous obtenons la suite de fonctions

$$\begin{aligned} &\theta_1(\alpha_{s_1}, \alpha_{s_2}, \ldots, \alpha_{s_m}), \\ &\theta_2(\alpha_{s_1}, \alpha_{s_2}, \ldots, \alpha_{s_m}), \\ &\cdots\cdots\cdots\cdots \\ &\theta_M(\alpha_{s_1}, \alpha_{s_2}, \ldots, \alpha_{s_m}). \end{aligned}$$

Choisissons deux cortèges disjoints $X' = \{\delta_j\}_{j=t+1}^{t+m}$ et $Y' = \{\beta_j\}_{j=1}^{M}$ et formons la suite de formules

$$\left.\begin{aligned} \beta_1 &:= \theta_1(\delta_{t+1}, \delta_{t+2}, \ldots, \delta_{t+m}); \\ \beta_2 &:= \theta_2(\delta_{t+1}, \delta_{t+2}, \ldots, \delta_{t+m}); \\ &\cdots\cdots\cdots\cdots \\ \beta_M &:= \theta_M(\delta_{t+1}, \delta_{t+2}, \ldots, \delta_{t+m}); \end{aligned}\right\} \quad (4.8)$$

Il est aisé de voir que la colonne de formules obtenue représente une suite normale avec le cortège d'entrée X' et le cortège de sortie Y'.

Désignons par G' l'ensemble d'états du cortège X' satisfaisant à la condition suivante: à tout état $\Phi' \subset G'$ du cortège X', il correspond de manière biunivoque un état $\Phi \subset G$ du cortège X et la relation

$$\Phi'(\delta_{t+j}) = \Phi(\alpha_{s_j}) \qquad (j = 1, 2, \ldots, m)$$

a lieu.

La suite normale (4.8) ayant X' pour cortège d'entrée, Y' pour cortège de sortie, et considérée avec G', représente un complexe que nous appellerons *équivalent* du complexe K et désignerons par KE.

Il est évident que le complexe K et son équivalent KE sont équivalents, i.e.

$$K = KE.$$

§ 4.3. Reconnaissance de l'équivalence de deux complexes uniformes d'après leurs équivalents

Soient K_1 et K_2 deux complexes uniformes. Supposons qu'un équivalent K_1E du complexe K_1 ait le domaine de définition G' et la suite normale

$$\left.\begin{array}{l} \beta_1 := \theta_1' (\delta_{t+1}, \delta_{t+2}, \ldots, \delta_{t+m}); \\ \beta_2 := \theta_2' (\delta_{t+1}, \delta_{t+2}, \ldots, \delta_{t+m}); \\ \cdots\cdots\cdots\cdots\cdots \\ \beta_{M'} := \theta_{M'}' (\delta_{t+1}), \delta_{t+2}, \ldots, \delta_{t+m}); \end{array}\right\} \quad (4.9)$$

avec X' pour cortège d'entrée et Y' pour cortège de sortie, et qu'un équivalent K_2E du complexe K_2 ait le domaine de définition G'' et la suite normale

$$\left.\begin{array}{l} \beta_1 := \theta_1'' (\delta_{\tau+1}, \delta_{\tau+2}, \ldots, \delta_{\tau+\mu}); \\ \beta_2 := \theta_2'' (\delta_{\tau+1}, \delta_{\tau+2}, \ldots, \delta_{\tau+\mu}); \\ \cdots\cdots\cdots\cdots\cdots \\ \beta_{M''} := \theta_{M''}'' (\delta_{\tau+1}, \delta_{\tau+2}, \ldots, \delta_{\tau+\mu}); \end{array}\right\} \quad (4.10)$$

avec X'' pour cortège d'entrée et Y'' pour cortège de sortie.

Supposons, de plus, que l'intersection $X' \cap X''$ des cortèges d'entrée n'est pas vide et que, pour tout état appartenant à l'un des ensembles G' et G'', il existe au moins un état dans l'autre ensemble qui coïncide avec lui sur $X' \cap X''$.

Désignons $X^* = X' \cup X''$, et soit G^* l'ensemble des états du cortège X^* qui sont liés aux états des cortèges X' et Y' de la façon suivante :

a) pour tout couple d'états dont le premier appartient à G', le deuxième à G'' et qui coïncident sur $X' \cap X''$, il existe dans l'ensemble G^* un état qui coïncide avec le premier sur X' et avec le deuxième sur X'';

b) pour tout état appartenant à G^* du cortège X^*, il existe un couple d'états dont le premier appartient à G', le deuxième à G'', et pour lesquels la condition a) est satisfaite.

Choisissons dans les seconds membres des formules de la suite normale (4.9) de l'équivalent K_1E toutes les fonctions non identi-

ques deux à deux θ'_{a_i}. De manière analogue, choisissons les fonctions θ''_{b_i} dans les seconds membres de la suite normale (4.10) de l'équivalent K_2E. En vertu des §§ 4.1 et 4.2, pour que les complexes uniformes K_1 et K_2 soient équivalents, il faut et il suffit que :

1. Les tailles des suites de fonctions choisies soient égales, c'est-à-dire que l'on obtienne deux suites:

$$\theta'_{a_1}, \theta'_{a_2}, \ldots, \theta'_{a_n}, \quad \theta''_{b_1}, \theta''_{b_2}, \ldots, \theta''_{b_n}.$$

2. Sur l'ensemble G^*, on ait les identités

$$\begin{aligned} \theta'_{a_1} &\equiv \theta''_{b_1}, \\ \theta'_{a_2} &\equiv \theta''_{b_2}, \\ &\cdots\cdots \\ \theta'_{a_n} &\equiv \theta''_{b_n}. \end{aligned} \tag{4.11}$$

Ainsi, à l'aide d'équivalents de complexes normaux uniformes, on est en mesure de reconnaître l'équivalence de tels complexes dès que l'on sait reconnaître les identités des fonctions (4.11). On sait que le problème de reconnaissance des identités est en général insoluble (il n'y a pas d'algorithme de reconnaissance de toutes les identités). Tout à fait de même, le problème de reconnaissance de l'équivalence de complexes uniformes est dans le cas général insoluble. Or, dans tous les cas qui se présentent en pratique, on arrive à le résoudre par la méthode décrite.

EXEMPLE 4.2. Entendons par cellules les lettres qui désignent certaines grandeurs, et par états de cellules, les valeurs de ces grandeurs (i.e. les nombres).

Considérons un complexe K_1 avec la suite normale

$$\begin{aligned} u &:= x + \sin^2 z + \cos^2 z; \\ v &:= y \cdot u; \\ w &:= v; \end{aligned}$$

le cortège d'entrée $X_1 = \{x, y, z\}$, le cortège de sortie $Y_1 = \{u, v, w\}$ et le domaine de définition G_1 qui se compose de tous les états du cortège X_1, c'est-à-dire est décrit par le système d'inégalités

$$\begin{aligned} -\infty < x < +\infty, \\ -\infty < y < +\infty, \\ -\infty < z < +\infty. \end{aligned}$$

Considérons encore un complexe K_2 avec la suite normale

$$u: = x + \frac{2}{\pi} (\text{arc } \sin t + \text{arc } \cos t);$$

$$v: = y \cdot u;$$

le cortège d'entrée $X_2 = \{x, y, t\}$, le cortège de sortie $Y_2 = \{u, v\}$ et le domaine de définition G_2 décrit par le système d'inégalités

$$-\infty < x < +\infty,$$

$$-\infty < y < +\infty,$$

$$0 \leqslant t \leqslant 1.$$

Montrons que $K_1 \sim K_2$.

En effet, après avoir exécuté l'algorithme de substitutions pour le complexe K_1, nous obtenons la suite de fonctions

$$x + \sin^2 z + \cos^2 z,$$

$$y\,(x + \sin^2 z + \cos^2 z),$$

$$y\,(x + \sin^2 z + \cos^2 z).$$

En y remplaçant les lettres x, y, z respectivement par δ_2, δ_3, δ_4, nous obtenons l'équivalent K_1E avec la suite normale

$$\beta_1: = \delta_2 + \sin^2 \delta_4 + \cos^2 \delta_4;$$

$$\beta_2: = \delta_3\,(\delta_2 + \sin^2 \delta_4 + \cos^2 \delta_4);$$

$$\beta_3: = \delta_3\,(\delta_2 + \sin^2 \delta_4 + \cos^2 \delta_4);$$

ayant $X'_1 = \{\delta_i\}_{i=2}^{4}$ pour cortège d'entrée. Le domaine de définition de K_1E peut être décrit par les inégalités

$$-\infty < \delta_2 < +\infty,$$

$$-\infty < \delta_3 < +\infty,$$

$$-\infty < \delta_4 < +\infty.$$

En effectuant les opérations analogues sur le complexe K_2 et remplaçant x, y, t respectivement par δ_2, δ_3, δ_1, nous aboutissons à l'équivalent K_2E du complexe K_2 avec la suite normale

$$\beta_1: = \delta_1 + \frac{2}{\pi} (\text{arc } \sin \delta_1 + \text{arc } \cos \delta_1);$$

$$\beta_2: = \delta_3 \left[\delta_2 + \frac{2}{\pi} (\text{arc } \sin \delta_1 + \text{arc } \cos \delta_1) \right];$$

ayant $X'_2 = \{\delta_i\}_{i=1}^{3}$ pour cortège d'entrée. Le domaine de définition de K_2E peut être décrit par les inégalités

$$0 \leqslant \delta_1 \leqslant 1,$$
$$-\infty < \delta_2 < +\infty,$$
$$-\infty < \delta_3 < +\infty.$$

Désignons par G^* l'ensemble des états du cortège $X^* = X'_1 \cup \cup X'_2 = \{\delta_i\}_{i=1}^{4}$ que l'on peut décrire par le système d'inégalités

$$0 \leqslant \delta_1 \leqslant 1,$$
$$-\infty < \delta_2 < +\infty,$$
$$-\infty < \delta_3 < +\infty,$$
$$-\infty < \delta_4 < +\infty.$$

Choisissons parmi les seconds membres des formules de K_1E, puis de K_2E les fonctions non identiques deux à deux et comparons les suites de fonctions ainsi obtenues. Evidemment, pour tout état de G^*, on a les identités

$$\delta_2 + \sin^2\delta_4 + \cos^2\delta_4 \equiv \delta_2 + \frac{2}{\pi}(\arcsin\delta_1 + \arccos\delta_1),$$

$$\delta_3(\delta_2 + \sin^2\delta_4 + \cos^2\delta_4) \equiv \delta_3\left[\delta_2 + \frac{2}{\pi}(\arcsin\delta_1 + \arccos\delta_1)\right],$$

puisque, pour les états de G^*, on a

$$\sin^2\delta_4 + \cos^2\delta_4 \equiv 1,$$
$$\arcsin\delta_1 + \arccos\delta_1 \equiv \frac{\pi}{2}.$$

Le résultat obtenu signifie que les complexes K_1 et K_2 sont équivalents, c'est-à-dire que $K_1 \sim K_2$.

§ 4.4. Système complet de transformations équivalentes de complexes uniformes

Si, en modifiant un complexe K, nous arrivons à un complexe K' équivalent à K, nous dirons qu'*il est permis* d'effectuer les modifications en question.

Désignons par le symbole Λ une formule vide, en convenant que chaque formule de la suite normale d'un complexe est précédée et suivie d'une formule vide. Si chacune des lettres A et B désigne une suite de formules, et s'il est permis de remplacer, dans la suite normale d'un complexe, la suite A par la suite B, nous noterons

$$A \to B.$$

L'ensemble de deux notations $A \to B$, $B \to A$ sera figuré comme

$$A \rightleftarrows B.$$

La notation

$$A \rightleftharpoons \Lambda \tag{4.12}$$

signifiera qu'il est permis de remplacer le groupe de formules A par une formule vide, autrement dit qu'on peut *supprimer* ou *introduire* ce groupe dans n'importe quel endroit de la suite normale du complexe. La notation de la forme (4.12) avec une cellule à la place de A sera employée lorsqu'il faut montrer qu'il est permis de supprimer la cellule dans le cortège ou de l'y inclure.

Convenons d'exprimer l'identité des cellules (lettres) α_{s_i} et α_j par la notation

$$\alpha_{s_i} (=) \alpha_j,$$

et leur différence par la notation

$$\alpha_{s_i} (\neq) \alpha_j.$$

T r a n s f o r m a t i o n C-1. Si l'identité $\varphi \equiv \psi$ a lieu, alors

$$[\alpha_{t_i} := \varphi;] \rightarrow [\alpha_{t_i} := \psi;]$$

T r a n s f o r m a t i o n A-2 et s o n i n v e r s e B-2:

$$[\alpha_j := \alpha_j] \rightleftharpoons \Lambda$$

Si α_j n'appartient pas au cortège d'entrée d'un complexe, on doit l'y inclure en effectuant la transformation B-2.

T r a n s f o r m a t i o n C-3. Si sont vérifiées les conditions $\alpha_{t_i} (\neq) \alpha_{t_{i+1}}$, $\alpha_{t_i} \neq sa\varphi_{i+1}$ et $\alpha_{t_{i+1}} \neq sa\varphi_i$, on a

$$\begin{bmatrix} \alpha_{t_i} := \varphi_i; \\ \alpha_{t_{i+1}} := \varphi_{i+1}; \end{bmatrix} \rightarrow \begin{bmatrix} \alpha_{t_{i+1}} := \varphi_{i+1}; \\ \alpha_{t_i} := \varphi_i; \end{bmatrix}.$$

T r a n s f o r m a t i o n A-4. Si les conditions $\alpha_{t_{i+1}} \neq sa\varphi_i$, $\alpha_{t_{i+1}} (\neq) \alpha_{t_i}$ sont remplies, on a

$$\begin{bmatrix} \alpha_{t_i} := \varphi_i; \\ \alpha_{t_{i+1}} := \varphi_{i+1} (\alpha_{t_i}, \alpha_{j \neq t_i}); \end{bmatrix} \rightarrow \begin{bmatrix} \alpha_{t_{i+1}} := \varphi_{i+1} (\varphi_i, \alpha_{j \neq t_i}); \\ \alpha_{t_i} := \varphi_i; \end{bmatrix}.$$

Ici nous écrivons $\varphi_{i+1} (\alpha_{t_i}, \alpha_{j \neq t_i})$ au lieu de $\varphi_{i+1} (\alpha_1, \alpha_2, \ldots, \alpha_n)$ pour mettre en relief l'argument α_{t_i} auquel on doit substituer φ_i lors de la transformation.

T r a n s f o r m a t i o n B-4. Si les conditions $\alpha_{t_i} \neq sa\varphi_{i+1}$ et $\alpha_{t_{i+1}} (\neq) \alpha_{t_i}$ sont remplies, alors

$$\begin{bmatrix} \alpha_{t_i} := \varphi_i (\varphi_{i+1}, \alpha_{j \neq t_{i+1}}); \\ \alpha_{t_{i+1}} := \varphi_{i+1}; \end{bmatrix} \rightarrow \begin{bmatrix} \alpha_{t_{i+1}} := \varphi_{i+1}; \\ \alpha_{t_i} := \varphi_i (\alpha_{t_{i+1}}, \alpha_{j \neq t_{i+1}}); \end{bmatrix}.$$

Appelons *éléments* d'un complexe son cortège d'entrée $\{\alpha_{s_i}\}_{i=1}^{m}$; les premiers membres des formules de sa suite normale $\alpha_{t_1}, \alpha_{t_2}, \ldots$ $\ldots, \alpha_{t_N}$; les deuxièmes membres de ces formules avec les signes

d'affectation qui les précèdent : $:= \varphi_1$; $:= \varphi_2$; . . . ; $:= \varphi_N$; son cortège de sortie $\{\alpha_{r_i}\}_{i=1}^{M}$. Ordonnons les éléments du complexe en leur attribuant les numéros de la manière suivante : 1 au cortège d'entrée ; $2i$ à l'élément $:= \varphi_i$; $2i+1$ à l'élément α_{t_i} ; $2N+2$ au cortège de sortie.

Soit α une lettre. Nous dirons que le premier élément du complexe est précédé d'une α-barrière ; si $\alpha_{t_i} (=) \alpha$, alors une α-barrière sépare les éléments $2i$ et $(2i+1)$ du complexe ; le dernier élément du complexe est suivi d'une α-barrière ; il n'y a pas d'autres α-barrières dans le complexe. Numérotons les α-barrières du complexe dans l'ordre de croissance des numéros des éléments du complexe.

L'ensemble des éléments du complexe qui se trouvent entre deux α-barrières successives sera appelé α-chaînon. Attribuons à tout α-chaînon le numéro de l'α-barrière immédiatement précédente.

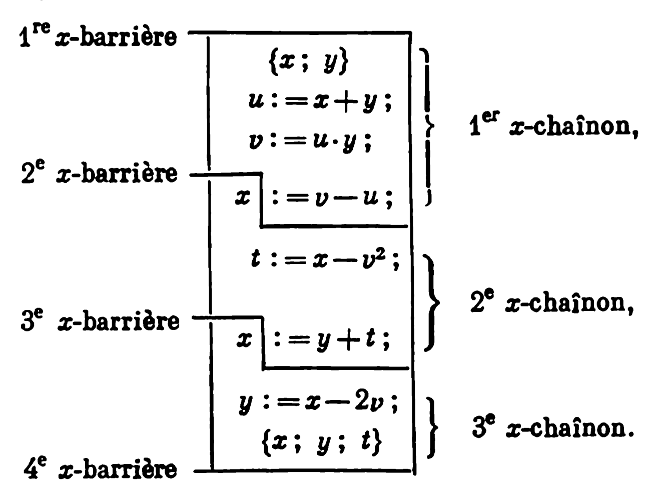
EXEMPLE 4.3. Considérons le complexe ayant les cortèges d'entrée et de sortie

$$X = \{x;\ y\};\quad Y = \{x;\ y;\ t\}$$

et la suite normale

$$\begin{aligned} u &:= x + y;\\ v &:= u \cdot y;\\ x &:= v - u;\\ t &:= x - v^2;\\ x &:= y + t;\\ y &:= x - 2v; \end{aligned}$$

Ce complexe K se compose des x-chaînons suivants :



Les éléments du complexe qui appartiennent à un α-chaînon seront appelés éléments de l'α-chaînon. Convenons d'ordonner les éléments d'un α-chaînon par ordre de croissance des numéros qu'on leur attribue en tant qu'aux éléments du complexe.

Tout ensemble d'éléments d'un α-chaînon qui est composé de son premier élément et de quelques formules suivant cet élément sera désigné par le symbole M_α.

Tout ensemble formé par quelques formules successives appartenant à un chaînon et par le dernier élément du chaînon qui les suit immédiatement sera désigné par le symbole N_α.

Supposons qu'un complexe K contient un α-chaînon et un β-chaînon qui satisfont aux conditions suivantes:

1) le premier élément de l'α-chaînon précède le premier élément du β-chaînon;

2) entre ces deux éléments, il y a une seule β-barrière et il n'y a aucune α-barrière.

Rejetons le premier élément du β-chaînon et trouvons l'intersection du reste du β-chaînon avec l'α-chaînon. Désignons l'ensemble trouvé par $O_{\alpha,\,\beta}$.

Soulignons que le premier élément de M_α est ou bien le cortège d'entrée X (qui peut contenir ou ne pas contenir une lettre α_{s_i} identique à α), ou bien une lettre α_{t_i} identique à α. Les premiers membres des autres formules faisant partie de M_α sont différents de α. Les premiers membres des formules appartenant à N_α sont tous différents de α, et ceux qui appartiennent à $O_{\alpha,\,\beta}$ diffèrent de α et de β.

Pour indiquer la présence éventuelle d'une lettre δ dans un élément d'un ensemble désigné par l'un des symboles M_α, N_α, $O_{\alpha,\,\beta}$ et $:= \varphi$, nous écrirons cette lettre δ entre parenthèses à droite du symbole correspondant. De même, pour montrer l'absence de la lettre δ dans un ensemble, nous mettrons entre parenthèses $\bar{\delta}$.

En effectuant une substitution de lettres dans les éléments des ensembles M_α, N_α, $O_{\alpha,\,\beta}$ ou $:= \varphi$ nous le ferons aussi à l'intérieur des parenthèses écrites à droite de chacun de ces symboles. Certaines substitutions peuvent entraîner une modification d'indice dans les symboles M_α, N_α, $O_{\alpha,\,\beta}$. Pour montrer qu'une lettre (par exemple, δ) est *partout ou dans certains endroits* remplacée par une autre lettre (par exemple, γ), nous écrirons deux fois la lettre remplacée entre parenthèses et nous remplacerons par la nouvelle lettre une seule des lettres répétées; par exemple, le remplacement partout ou dans certains endroits de la lettre δ par la lettre γ dans les éléments de l'ensemble $O_{\alpha,\,\beta}$ sera noté comme suit: nous écrivons initialement $O_{\alpha,\,\beta}(\delta, \delta)$, et $O_{\alpha,\,\beta}(\delta, \gamma)$ ou $O_{\alpha,\,\beta}(\gamma, \delta)$ après le remplacement.

Transformation A-5 et son inverse B-5:

$$\begin{bmatrix} M_\alpha(\alpha, \bar{\beta}) \\ \beta := \varphi(\alpha, \bar{\beta}); \\ N_\alpha(\bar{\alpha}) \end{bmatrix} \rightleftarrows \begin{bmatrix} M_\beta(\beta, \bar{\alpha}) \\ \beta := \varphi(\beta, \bar{\alpha}); \\ N_\alpha(\bar{\alpha}) \end{bmatrix}.$$

Transformation A-6 et son inverse B-6:

$$\begin{bmatrix} \alpha := \varphi; \\ N_\alpha(\bar{\alpha}) \end{bmatrix} \rightleftarrows [N_\alpha(\bar{\alpha})].$$

Transformation A-7 et son inverse B-7:

$$\begin{bmatrix} \alpha := \varphi(\bar{\alpha}); \\ \beta := \varphi(\bar{\alpha}); \\ O_{\alpha,\beta}(\alpha, \alpha, \beta) \end{bmatrix} \rightleftarrows \begin{bmatrix} \alpha := \varphi(\bar{\alpha}); \\ \beta := \varphi(\bar{\alpha}); \\ O_{\alpha,\beta}(\alpha, \beta, \beta) \end{bmatrix}.$$

Si une transformation A-7 appliquée à un complexe pour lequel $\alpha \in Y$ et $\beta \in Y$ fait apparaître la lettre β deux fois dans le cortège Y (donc α disparaît), alors l'une des copies de β est à supprimer dans Y.

Si la transformation B-7 touche le cortège de sortie Y d'un complexe, et $\beta \in Y$, alors, avant d'exécuter B-7, on peut inclure dans Y un « deuxième exemplaire » de β pour le remplacer par α, lors de la transformation.

EXEMPLE 4.4. Considérons le complexe

$\{x;\ y\}$
$x := x + 5;$
$v := u - y;$
$z := v - u;$
x | $:= v + u:$
$\{x;\ z\}$

On peut appliquer la transformation A-5 à son premier x-chaînon:

$$\begin{bmatrix} M_x(x, \bar{v}) \\ v := u - y; \\ N_x(\bar{x}) \end{bmatrix} \rightarrow \begin{bmatrix} M_v(v, \bar{x}) \\ v := v - y; \\ N_x(\bar{x}) \end{bmatrix},$$

où $v := \varphi(x, \bar{v})$; est réalisé comme $v := u - y$; .On aboutit au complexe

$$\{v;\ y\}$$
$$u := v + 5;|$$
$$v := u - y;$$
$$z := v - y:$$
$$|x := v + u,$$
$$\{x;\ z\}.$$

Transformation A-8 et son inverse B-8. Si sont remplies les conditions $\alpha_{s_j} \neq s\alpha\varphi_i$ pour $i = 1, 2, \ldots, N$ et $\alpha_{s_j} \notin Y$ alors

$$[\alpha_{s_j}] \rightleftarrows [\Lambda],$$

c'est-à-dire, dans les conditions indiquées, on peut supprimer une lettre dans le cortège X ou l'y inclure.

Transformation A-9 et son inverse B-9. Si la suite normale d'un complexe se termine par la formule $\alpha_{r_k} := \alpha_r$ et $\alpha_{r_l} \in Y$, alors

$$[\alpha_{r_k}] \rightleftarrows [\Lambda],$$

c'est-à-dire, dans les conditions formulées, la lettre α_{r_k} peut être supprimée ou introduite dans le cortège Y.

Les seize transformations équivalentes signalées forment un *système complet* en ce sens qu' étant donné deux complexes équivalents, on peut réduire l'un à l'autre au moyen d'un nombre fini des transformations du système. Les transformations suivantes découlent des précédentes:

$$\begin{bmatrix} \alpha_k := \varphi_l; \\ \alpha_k := \varphi_{l+1}(\alpha_k, \alpha_{j \neq k}); \end{bmatrix} \leftrightarrows [\alpha_k := \varphi_{l+1}(\varphi_l, \alpha_{j \neq k});].$$

Si θ est la suite normale d'un complexe K, et si les états du cortège d'entrée X qui appartiennent au domaine de définition du complexe satisfont à la relation $\alpha_j \equiv \psi(\alpha_1, \alpha_2, \ldots, \alpha_n)$, alors

$$[\theta] \rightleftarrows [\alpha_j \underset{\theta}{:}= \psi;],$$

c'est-à-dire qu'il est permis de faire précéder la suite normale de la formule $\alpha_j = \psi$; ou de la supprimer si elle est la première formule de la suite normale.

§ 4.5. Opérations sur les complexes uniformes. Paquets de complexes uniformes

Les transformations équivalentes des complexes uniformes qui sont, au fond, des transformations d'opérateurs non logiques, s'utilisent, naturellement, pour les transformations équivalentes d'algorithmes plus compliqués. Ceci vu, il nous faut étudier certaines opérations sur les complexes uniformes dont nous aurons besoin par la suite.

4.5.1. Addition des complexes uniformes. Soit deux complexes uniformes K_1 et K_2 qui ne diffèrent (éventuellement) que par leurs domaines de définition G_1 et G_2, leurs cortèges d'entrée, les suites normales et les cortèges de sortie étant graphiquement identiques. Alors on appelle *somme faible* des complexes K_1 et K_2 le complexe uniforme K qui ne diffère de chacun des complexes K_1, K_2 que (peut-être) par son domaine de définition G, où

$$G = G_1 + G_2.$$

L'opération qui fait correspondre à deux complexes K_1 et K_2 leur somme faible s'appelle *addition faible* des complexes et se désigne par le signe « $+$ ». On écrit donc

$$| \ K = K_1 + K_2.$$

Soient K_1 et K_2 deux complexes uniformes, et supposons qu'il existe un complexe uniforme K tel que $K = K_1' + K_2'$, où $K_1' = K_1$, $K_2' = K_2$. Le complexe K s'appelle *somme* des complexes K_1 et K_2, et l'on écrit toujours

$$K = K_1 + K_2.$$

La somme faible de deux complexes est un cas particulier de leur somme.

L'opération qui à deux complexes uniformes K_1 et K_2 fait correspondre leur somme s'appelle *addition* et les complexes K_1 et K_2 sont *termes*.

L'addition des complexes uniformes est *commutative*

$$K_1 + K_2 = K_2 + K_1$$

et *associative*

$$K_1 + (K_2 + K_3) = (K_1 + K_2) + K_3.$$

La dernière propriété permet de supprimer les parenthèses dans les sommes polynomiales; par exemple, il est possible d'écrire

$$K_1 + K_2 + K_3.$$

Etant donné un complexe uniforme K avec G pour domaine de définition, on peut partager G en deux parties disjointes G_1 et G_2, de sorte que

$$G = G_1 + G_2$$

et considérer deux complexes K_1 et K_2, avec respectivement G_1 et G_2 pour domaines de définition, dont chacun ne diffère de K que par son domaine de définition. On aura évidemment

$$K = K_1 + K_2.$$

L'opération décrite, qui fait correspondre à tout complexe uniforme K deux complexes K_1 et K_2 dont il est la somme, s'appelle *décomposition du complexe K en termes*. Comme il existe plusieurs façons de partager le domaine de définition G du complexe K en deux parties disjointes, la décomposition du complexe uniforme K en deux termes n'est pas univoque.

4.5.2. Multiplication des complexes uniformes. Soient K_1 et K_2 deux complexes ayant les suites normales θ_1, θ_2, les cortèges d'entrée X_1, X_2, les cortèges de sortie Y_1, Y_2, les cortèges intermédiaires Z_1, Z_2 et les domaines de définitions G_1, G_2 respectivement. Supposons que l'intersection $X_2 \cap Z_1$ soit vide. Désignons

$$X = X_1 + (X_2 - Y_1),\ Y = Y_2 + (Y_1 - Z_2).$$

Il est aisé de voir que la colonne de formules que l'on obtient en écrivant la suite normale θ_2 au-dessous de la suite normale θ_1 représente une nouvelle suite normale θ avec X pour cortège d'entrée et Y pour cortège de sortie.

On obtient ainsi un complexe K que nous appelons *produit* des complexes K_1 et K_2 et notons K_1K_2.

L'opération qui fait correspondre à deux complexes uniformes leur produit s'appelle *multiplication* des complexes.

La condition imposée plus haut aux cortèges des complexes K_1 et K_2 veut dire que les résultats intermédiaires de l'opérateur A_1 représenté par le complexe K_1 ne sont pas les données initiales de l'opérateur A_2 correspondant au complexe K_2. Remarquons que l'opérateur A qui correspond au complexe K_1K_2 est le produit des opérateurs A_1 et A_2.

On peut considérer le produit de plusieurs (plus que deux) complexes, si chaque complexe et chacun des complexes qui le suit (immédiatement ou non) satisfont à la condition imposée aux cortèges.

La multiplication des cortèges uniformes est *associative*

$$K_1\,(K_2K_3) = (K_1K_2)\,K_3.$$

Cela permet de supprimer les parenthèses dans les produits de plusieurs facteurs. On peut, par exemple, écrire

$$K_1K_2K_3.$$

La multiplication des complexes uniformes *n'est pas commutative*.

Soit K un complexe uniforme de suite normale θ. On peut partager la suite normale θ (par ordre de croissance des numéros des formules) en parties $\theta_1, \theta_2, \ldots, \theta_k$ et considérer les complexes $K_1, K_2, \ldots, K_k$ avec les suites normales $\theta_1, \theta_2, \ldots, \theta_k$ et les cortèges d'entrée et de sortie dûment choisis. Dans certains cas on doit effectuer préalablement une transformation simple du complexe K pour satisfaire les conditions imposées aux cortèges des complexes formés. On aura alors

$$K = K_1K_2 \ldots K_k.$$

Le procédé décrit sera appelé *factorisation du complexe uniforme K*. Il est évident que la factorisation du complexe n'est pas univoque.

4.5.3. Complexes unitaires et complexes inverses. Soit K un complexe uniforme, G, son domaine de définition et $X = Y$. Si dans K à tout état appartenant à G du cortège X correspond le même état (du cortège Y), on dit que le complexe K est *unitaire* et l'on note

$$K = \Lambda.$$

Il est aisé de voir que si $K = \Lambda$, alors $K = K'$, où K' et K ont le même domaine de définition G et la même suite normale

$$\alpha_{s_i} = \alpha_{s_i} \quad (i = 1, 2, \ldots, m).$$

Soit K_n un complexe uniforme quelconque. S'il existe un complexe K_{-n} tel que

$$K_nK_{-n} = \Lambda,$$

alors K_{-n} s'appelle *inverse à droite* de K_n, et K_n *inverse à gauche* de K_{-n}.

Si le complexe K_{-n} est, par rapport au complexe K_n, inverse à gauche et à droite à la fois, c'est-à-dire si

$$K_nK_{-n} = K_{-n}K_n = \Lambda,$$

alors le complexe K_{-n} s'appelle inverse de K_n.

4.5.4. Paquets de complexes uniformes. Equivalence des paquets. Soit $K_1, K_2, \ldots, K_l, \ldots$ un ensemble fini ou dénombrable de complexes avec le cortège d'entrée commun X et les domaines de définition deux à deux disjoints $G_1, G_2, \ldots, G_l$. Désignons

$$G = G_1 + G_2 + \ldots + G_l + \ldots$$

L'ensemble de complexes K_1, K_2, . . ., K_l, . . . s'appelle *paquet de complexes* de cortège d'entrée X et de domaine de définition G.

Deux paquets de complexes sont dits *équivalents* si à l'aide des opérations de décomposition et d'addition des complexes, on peut transformer ces paquets de façon à faire correspondre à chaque complexe du premier paquet un complexe équivalent du second.

§ 4.6. Transformations équivalentes principales des algorithmes donnés en YALS

Dans ce qui précède, nous avons considéré les transformations équivalentes des complexes uniformes, c'est-à-dire, au fond, des opérateurs. A présent, nous passons aux transformations équivalentes des algorithmes arbitraires qui comprennent les opérateurs I_0, T_v, les opérateurs d'action, de variation et logiques et qui ne comprennent pas de signes de passage ouvrants dépendant de paramètres.

On a déjà dit qu'en étudiant l'équivalence, il faut considérer un algorithme donné en YALS avec ses données initiales et ses résultats. Or, dans le cas général, il est impossible d'indiquer les données initiales en donnant un ensemble fixe X des cellules initiales, come c'était le cas pour les complexes uniformes. Le fait qu'un algorithme contient des opérateurs dépendant de paramètres entraîne que pour ranger les données initiales on a besoin d'un nombre différent de cellules. Il en est de même des résultats. Il existe pourtant une méthode d'étude de l'équivalence des algorithmes arbitraires en YALS qui permet de franchir les difficultés liées à la nécessité d'indiquer les ensembles des données initiales et des résultats. Avant de la décrire, convenons d'employer, à côté du terme « transformation équivalente d'un algorithme », le terme « transformation équivalente d'un schéma logique (ou d'un schéma tout court) ».

Au cours de l'exécution d'un schéma logique, on peut écrire la suite normale d'un opérateur d'action chaque fois qu'on le réalise. On obtiendra en fin des comptes une suite normale avec un cortège d'entrée et un cortège de sortie bien déterminés, c'est-à-dire qu'on aboutira à un complexe uniforme.

Un tel complexe uniforme sera appelé *réalisation* de l'algorithme (du schéma logique) donné en YALS. Le domaine de définition d'un schéma est la somme des domaines de définition de ses réalisations qui diffèrent l'une de l'autre par les cellules qui leur sont attribuées et par les états des cellules. Ainsi, à chaque schéma (algorithme) il correspond un paquet de réalisations.

Il est évident que pour que deux algorithmes en YALS (deux schémas) soient équivalents, il faut et il suffit que les paquets de leurs réalisations soient équivalents.

Il est aisé de voir que l'ensemble des réalisations d'un schéma est au plus dénombrable.

Supposons qu'un schéma $\sum$ satisfait à la condition suivante: pour tout signe de passage ouvrant du schéma, le signe de passage fermant qui lui correspond est situé plus à droite.

Un tel schéma sera dit *direct*.

Il est évident que l'ensemble des réalisations d'un schéma direct est fini.

Deux cas des transformations équivalentes des schémas se présentent: les transformations locales, où il s'agit de remplacer une expression par une autre expression, et les transformations globales, où il faut remplacer, en concordance, plusieurs expressions qui se trouvent dans les différents endroits du schéma. Dans le premier cas il s'agit de l'équivalence des expressions, dans le deuxième, de l'équivalence des schémas.

Pour indiquer que les expressions W_1, W_2, . . ., W_n occupent certaines places dans un schéma $\sum$, nous écrirons $\sum\{W_1, W_2, \ldots, W_n\}$. Le résultat de remplacement dans le dernier schéma des expressions W_1, W_2, . . ., W_n par les expressions respectives W'_1, W'_2, ..., W'_n sera désigné par $\sum\{W'_1, W'_2, \ldots, W'_n\}$.

Deux expressions W_1 et W_2 s'appellent équivalentes, si le fait que $\sum\{W_1\}$ est un schéma implique deux faits suivants:

1) $\sum\{W_2\}$ est encore un schéma;

2) ces deux schémas sont équivalents.

L'équivalence de deux expressions sera désignée par le signe d'égalité.

Les formules globales auront la forme

$$\sum\{W_1, W_2, \ldots, W_n\} = \sum\{W'_1, W'_2, \ldots, W'_n\}.$$

Le symbole Λ désignera une expression vide. On convient qu'une expression vide est présente implicitement après chaque expression élémentaire d'un schéma.

§ 4.7. Transformations équivalentes indépendantes des propriétés intrinsèques des opérateurs

Les transformations équivalentes qui suivent concernent l'algorithme d'exécution des schémas.

Il est permis de permuter les signes de passage fermants qui voisinent:

$$\lrcorner_{i} \lrcorner_{j} = \lrcorner_{i} \lrcorner_{j}$$

Si un schéma ne contient pas de signe de passage ouvrant d'indice k, il est permis de supprimer le signe de passage fermant de même indice :

$$\underset{k}{\lrcorner} = \Lambda. \tag{4.13}$$

Il est permis de permuter les signes de passage ouvrants (inférieur et supérieur) qui suivent immédiatement un opérateur logique :

$$P \overset{j}{\ulcorner} \underset{i}{\lrcorner} = P \underset{i}{\llcorner} \overset{j}{\ulcorner}.$$

La dernière formule permet d'écrire $\overset{j}{\underset{i}{\sqsubset}}$ à la place de $\overset{j}{\ulcorner} \underset{i}{\llcorner}$. Ainsi :

$$P \overset{j}{\ulcorner} \underset{i}{\llcorner} = P \underset{i}{\llcorner} \overset{j}{\ulcorner} = P \overset{j}{\underset{i}{\sqsubset}}.$$

Les signes $\underset{i}{\llcorner}$, $\overset{i}{\ulcorner}$ seront indifféremment désignés par λ_i, et une expression élémentaire contenant λ_i, par $R\lambda_i$, en sous-entendant par R l'ensemble de tous les symboles de l'expression élémentaire sauf λ_i.

Si le schéma $\sum\{R\lambda_i, \underset{i}{\lrcorner}\}$ ne contient pas de signes de passage d'indice j, alors

$$\sum\{R\lambda_i, \underset{i}{\lrcorner}\} = \sum\{R\lambda_j, \underset{j}{\lrcorner} \underset{i}{\lrcorner}\}.$$

Pour simplifier la notation des schémas logiques, on convient que

$$R\lambda_i \underset{i}{\lrcorner} = R \underset{i}{\lrcorner}. \tag{4.14}$$

Si, de plus, un schéma ne contient pas de signes de passage ouvrants d'indice i autres que λ_i, alors, en vertu de (4.13), on a :

$$R\lambda_i \underset{i}{\lrcorner} = R. \tag{4.15}$$

La convention (4.14) entraîne la validité des formules suivantes :

$$D \underset{i}{\llcorner} \underset{i}{\lrcorner} = D \underset{i}{\lrcorner},$$

$$V \underset{i}{\llcorner} \underset{i}{\lrcorner} = V \underset{i}{\lrcorner}.$$

$$P \overset{j}{\underset{i}{\sqsubset}} \underset{i}{\lrcorner} = P \ulcorner \underset{i}{\lrcorner},$$

$$P \overset{j}{\underset{i}{\sqsubset}} \underset{j}{\lrcorner} = P \underset{i}{\llcorner} \underset{j}{\lrcorner}.$$

Exemple 4.5. Les formules données plus haut simplifient remarquablement la notation. Ainsi, à la place de

$$I_0 \underset{1}{\llcorner} \underset{1}{\lrcorner} D_1 \underset{2}{\llcorner} \underset{2}{\lrcorner} P_1 \overset{4}{\ulcorner} \underset{3}{\llcorner} \underset{4}{\lrcorner} D_2 \underset{5}{\llcorner} \underset{3}{\lrcorner} \underset{5}{\lrcorner} D_3 \underset{6}{\llcorner} \underset{6}{\lrcorner} T$$

on peut ecrire

$$I_0 D_1 P_1 \underset{3}{\llcorner} D_2 \underset{3}{\lrcorner} D_3 T.$$

Supposons que les symboles S_1 et S_2 désignent la fermeture d'opérateurs. Alors

$$\sum\{S_1,\ S_2\} = \sum\{S_1,\ S_2,\ \Lambda\}. \tag{4.16}$$

Considérons le schéma $\sum\{S\ W\}$, où S est la fermeture d'un opérateur et W une expression quelconque. Si tous les indices des signes de passage fermants qui figurent dans SW diffèrent des indices des signes de passage ouvrants présents dans le reste du schéma $\sum\{SW\}$, alors

$$SW = \Lambda. \tag{4.17}$$

La dernière formule est aussi bien valable dans le cas où W représente une expression vide.

Soit w une expression élémentaire faisant partie de l'expression SW de la formule (4.17) et n'étant pas un signe de passage fermant. Alors

$$w = \Lambda.$$

On déduit de (4.17) les formules suivantes:

$$D \underset{i}{\llcorner} Q \underset{k}{\llcorner} = D \underset{i}{\llcorner}, \qquad P \overset{j}{\underset{i}{\sqsubset}} Q \underset{h}{\llcorner} = P \overset{j}{\underset{i}{\sqsubset}},$$

$$D \underset{i}{\llcorner} P \overset{l}{\underset{k}{\sqsubset}} = D \underset{i}{\llcorner}, \qquad P \overset{j}{\underset{i}{\sqsubset}} P_1 \overset{l}{\underset{k}{\sqsubset}} = P \overset{j}{\underset{i}{\sqsubset}},$$

$$D \underset{i}{\llcorner} T = D \underset{i}{\llcorner}, \qquad P \overset{j}{\underset{i}{\sqsubset}} T = P \overset{j}{\underset{i}{\sqsubset}},$$

$$V \underset{i}{\llcorner} Q \underset{h}{\llcorner} = V \underset{i}{\llcorner}, \qquad TQ \underset{k}{\llcorner} = T,$$

$$V \underset{i}{\llcorner} P \overset{l}{\underset{k}{\sqsubset}} = V \underset{i}{\llcorner}, \qquad TP \overset{l}{\underset{k}{\sqsubset}} = T,$$

$$V \underset{i}{\llcorner} T = V \underset{i}{\llcorner}, \qquad TT_1 = T.$$

Exemple 4.6. Considérons le schéma

$$\sum = I_0 D_1 \underset{1}{\llcorner} \underset{2}{\lrcorner} D_2 \underset{3}{\llcorner} \underset{1}{\lrcorner} P_3 \overset{5}{\underset{4}{\sqsubset}} \underset{3}{\lrcorner} D_6 \underset{2}{\llcorner} \underset{4}{\lrcorner} D_7 \underset{5}{\lrcorner} D_8 T_9.$$

Permutons $\underset{1}{\lrcorner}\, P_3 \overset{5}{\underset{4}{\sqsubset}}$ (fermeture de l'opérateur P_3) et $\underset{3}{\lrcorner}\, D_6 \underset{2}{\llcorner}$ (fermeture de l'opérateur D_6) d'après (4.16). Nous aurons

$$\sum = I_0 D_1 \underset{1}{\llcorner} \underset{2}{\lrcorner} D_2 \underset{3}{\llcorner} \underset{3}{\lrcorner} D_6 \underset{2}{\llcorner} \underset{1}{\lrcorner} P_3 \overset{5}{\underset{4}{\sqsubset}} \underset{4}{\lrcorner} D_7 \underset{5}{\lrcorner} D_8 T_9.$$

Puisque, dans ce schéma, $D_2 \underset{3}{\llcorner}\underset{3}{\lrcorner} = D_2$ et $P_3 \overset{5}{\underset{4}{\sqsubset}}\underset{4}{\lrcorner} = P_3 \overset{5}{\ulcorner}$, on peut écrire

$$\sum = I_0 D_1 \underset{1}{\llcorner}\underset{2}{\lrcorner} D_2 D_6 \underset{2}{\llcorner}\underset{1}{\lrcorner} P_3 \overset{5}{\ulcorner} D_7 \underset{5}{\lrcorner} D_8 T_9.$$

Ici $\underset{2}{\lrcorner} D_2 D_6 \underset{2}{\llcorner}$ représente l'expression SW indiquée dans (4.17). Elle ne contient aucun signe de passage fermant d'indice égal à celui d'un signe de passage ouvrant se trouvant dans le schéma en dehors de l'expression. Par conséquent, $\underset{2}{\lrcorner} D_2 D_6 \underset{2}{\llcorner} = \Lambda$, d'où

$$\sum = I_0 D_1 \underset{1}{\llcorner}\underset{1}{\lrcorner} P_3 \overset{5}{\ulcorner} D_7 \underset{5}{\lrcorner} D_8 T_9.$$

Compte tenu de $D_1 \underset{1}{\llcorner}\underset{1}{\lrcorner} = D_1$, on obtient définitivement

$$\sum = I_0 D_1 P_3 \overset{5}{\ulcorner} D_7 \underset{5}{\lrcorner} D_8 T_9.$$

Considérons une expression W et supposons, pour fixer les idées, que tous les signes de passage y sont explicites.

Supposons encore que W satisfait aux conditions:

1) si un signe de passage ouvrant extérieur d'indice α (voir p. 2.1.9) appartient à un opérateur S et W, alors le même signe de passage (de même indice) appartient à chaque opérateur de W identique à S ;

2) si λ_α et λ_β sont des signes de passage ouvrants de même type, intérieurs pour W et appartenant à des opérateurs identiques de W, alors les signes de passage fermants $\underset{\alpha}{\lrcorner}$ et $\underset{\beta}{\lrcorner}$ appartiennent eux aussi à des opérateurs identiques de W (ou à un même opérateur).

Une expression W qui satisfait à ces deux conditions s'appelle *parfaite*. Ses opérateurs identiques sont dits *semblables*. Les expressions élémentaires et les fermetures d'opérateurs sont des cas particuliers des expressions parfaites.

Une expression parfaite dans laquelle tout opérateur entre avec $n - 1$ opérateurs semblables est dite n-uple.

Exemple 4.7. L'expression

$$\underset{1}{\lrcorner}\, P_1 \overset{11}{\underset{7}{\sqsubset}} \underset{2}{\lrcorner}\, P_1 \overset{11}{\underset{6}{\sqsubset}} \underset{7}{\lrcorner} \underset{6}{\lrcorner}\, D_1 \underset{10}{\llcorner}$$

est parfaite. Ici $\overset{11}{\ulcorner}$ et $\underset{10}{\llcorner}$ sont des signes de passage ouvrants extérieurs, et $\underset{7}{\llcorner}$ et $\underset{6}{\llcorner}$, intérieurs.

EXEMPLE 4.8. L'expression

$$\underset{4}{\lrcorner} D_1 \underset{2}{\llcorner} \underset{3}{\lrcorner} \underset{1}{\lrcorner} P_1 \overset{10}{\underset{4}{\sqsubset}} \underset{5}{\lrcorner} D_1 \underset{3}{\llcorner} \underset{2}{\lrcorner} P_1 \overset{10}{\underset{5}{\sqsubset}}$$

est double. Ses opérateurs D_1 et P_1 ont chacun un opérateur semblable. Les signes de passage ouvrants supérieurs $\overset{10}{\ulcorner}$ qui appartiennent aux opérateurs P_1 sont extérieurs, tous les autres signes de passage ouvrants étant intérieurs.

Tout signe de passage fermant qui appartient à un opérateur Q d'une expression parfaite peut être transporté et placé devant n'importe quel opérateur Q' semblable à l'opérateur Q.

Désignons par π l'ensemble des signes de passage fermants qui appartiennent à la fermeture d'un opérateur.

Les formules suivantes résultent de la règle que nous venons de formuler:

$$\left.\begin{aligned} &\sum\{\underset{k}{\lrcorner} Q \underset{i}{\llcorner},\ \pi Q \underset{i}{\llcorner}\} = \sum\{Q \underset{i}{\llcorner},\ \underset{k}{\lrcorner} \pi Q \underset{i}{\llcorner}\},\\ &\sum\{\underset{k}{\lrcorner} P \overset{j}{\underset{i}{\sqsubset}},\ \pi P \overset{j}{\underset{i}{\sqsubset}}\} = \sum\{P \overset{j}{\underset{i}{\sqsubset}},\ \underset{k}{\lrcorner} \pi P \overset{j}{\underset{i}{\sqsubset}}\},\end{aligned}\right\} \qquad (4.18)$$

où Q est un opérateur non logique, P un opérateur logique. Les dernières formules impliquent à leur tour

$$\left.\begin{aligned} &\underset{k}{\lrcorner}\underset{l}{\lrcorner} Q \underset{i}{\llcorner} = \underset{k}{\lrcorner} Q \underset{i}{\llcorner} \underset{l}{\lrcorner} Q \underset{i}{\llcorner},\\ &\underset{k}{\lrcorner}\underset{l}{\lrcorner} P \overset{j}{\underset{i}{\sqsubset}} = \underset{k}{\lrcorner} P \overset{j}{\underset{i}{\sqsubset}} \underset{l}{\lrcorner} P \overset{j}{\underset{i}{\sqsubset}}.\end{aligned}\right\} \qquad (4.19)$$

Les formules (4.18) sont aussi bien valables dans le cas où Q est un opérateur terminal. Elles ont alors la forme

$$\sum\{\underset{k}{\lrcorner} T_\nu,\ \pi T_\mu\} = \sum\{T_\nu,\ \underset{k}{\lrcorner} \pi T_\mu\};$$

la première formule (4.19) s'écrit comme

$$\underset{k}{\lrcorner}\underset{l}{\lrcorner} T = \underset{k}{\lrcorner} T \underset{l}{\lrcorner} T.$$

Signalons encore les règles suivantes.

Si un schéma $\sum$ ne contient aucun signe de passage d'indice k, alors n'importe quel indice i des signes de passage du schéma peut être remplacé par k.

Si un schéma contient des signes de passage d'indices k et l, alors il est permis de remplacer tous les indices k par l et, en même temps, tous les indices l par k.

§ 4.8. Transformations principales des opérateurs logiques

La classe des formules qui ne sont pas liées aux propriétés intrinsèques des opérateurs contient encore deux formules:

$$\left.\begin{aligned} &\sum\{\pi P \overset{k}{\underset{k}{\sqsubset}},\ \underset{k}{\lrcorner}\} = \sum\{\Lambda,\ \pi \underset{k}{\lrcorner}\},\\ &P \overset{k}{\underset{k}{\sqsubset}} \underset{k}{\lrcorner} = \underset{k}{\lrcorner}. \end{aligned}\right\} \quad (4.20)$$

Nous désignerons par 0 ou 1 le prédicat constant et l'opérateur logique qui lui correspond, selon la valeur du prédicat. On peut également considérer comme constant un prédicat qui conserve sa valeur au cours de l'exécution de l'opérateur logique qui lui correspond, pour tout $g(x)$ de G. Pour les prédicats constants on a

$$0 \overset{j}{\underset{i}{\sqsubset}} = 0 \overset{i}{\underset{i}{\sqsubset}},$$

$$1 \overset{j}{\underset{i}{\sqsubset}} = 1 \overset{j}{\underset{j}{\sqsubset}}.$$

Les dernières formules, avec (4.20), fournissent

$$0 \overset{j}{\underset{i}{\sqsubset}} \underset{i}{\lrcorner} = \underset{i}{\lrcorner},$$

$$1 \overset{j}{\underset{i}{\sqsubset}} \underset{j}{\lrcorner} = \underset{j}{\lrcorner},$$

d'où

$$0 \overset{j}{\underset{i}{\sqsubset}} = P \overset{i}{\underset{i}{\sqsubset}},$$

$$1 \overset{j}{\underset{i}{\sqsubset}} = P \overset{j}{\underset{j}{\sqsubset}}.$$

Si les prédicats P_1 et P_2 ont les mêmes arguments essentiels et prennent des valeurs égales pour les mêmes combinaisons des valeurs des arguments essentiels, i.e. $P_1 \equiv P_2$, alors

$$P_1 \overset{j}{\underset{i}{\sqsubset}} = P_2 \overset{j}{\underset{i}{\sqsubset}}.$$

En particulier, cette égalité a lieu lorsque les prédicats P_1 et P_2 sont logiquement équivalents.

Les trois formules qui suivent permettent de décomposer (dans le cas d'opérateurs correspondant aux prédicats polyadiques) et de réunir les opérateurs logiques:

$$\bar{P} \overset{j}{\underset{i}{\sqsubset}} = P \overset{i}{\underset{j}{\sqsubset}},$$

$$P_1 \wedge P_2 \overset{j}{\underset{i}{\sqsubset}} = P_1 \underset{i}{\llcorner} P_2 \overset{j}{\underset{i}{\sqsubset}},$$

$$P_1 \vee P_2 \overset{j}{\underset{i}{\sqsubset}} = P_1 \overset{j}{\ulcorner} P_2 \overset{j}{\underset{i}{\sqsubset}}.$$

D'une grande importance sont les formules

$$\sum\{P \overset{j}{\underset{i}{\sqsubset}},\ \underset{i}{\lrcorner} P \overset{l}{\underset{k}{\sqsubset}}\} = \sum\{P \overset{j}{\underset{k}{\sqsubset}},\ \underset{i}{\lrcorner} P \overset{l}{\underset{k}{\sqsubset}}\},$$

$$\sum\{P \overset{j}{\underset{i}{\sqsubset}},\ \underset{j}{\lrcorner} P \overset{l}{\underset{k}{\sqsubset}}\} = \sum\{P \overset{l}{\underset{i}{\sqsubset}},\ \underset{j}{\lrcorner} P \overset{l}{\underset{k}{\sqsubset}}\},$$

qui impliquent les formules

$$P \overset{j}{\underset{i}{\sqsubset}} W \underset{i}{\lrcorner} P \overset{l}{\underset{h}{\sqsubset}} = P \overset{j}{\underset{k}{\sqsubset}} W \underset{i}{\lrcorner} P \overset{l}{\underset{k}{\sqsubset}},$$

$$P \overset{j}{\underset{i}{\sqsubset}} W \underset{j}{\lrcorner} P \overset{l}{\underset{k}{\sqsubset}} = P \overset{l}{\underset{i}{\sqsubset}} W \underset{j}{\lrcorner} P \overset{l}{\underset{k}{\sqsubset}},$$

$$\underset{i}{\lrcorner} P \overset{l}{\underset{k}{\sqsubset}} W P \overset{j}{\underset{i}{\sqsubset}} = \underset{i}{\lrcorner} P \overset{l}{\underset{k}{\sqsubset}} W P \overset{j}{\underset{k}{\sqsubset}},$$

$$\underset{j}{\lrcorner} P \overset{l}{\underset{k}{\sqsubset}} W P \overset{j}{\underset{i}{\sqsubset}} = \underset{j}{\lrcorner} P \overset{l}{\underset{k}{\sqsubset}} W P \overset{l}{\underset{i}{\sqsubset}}.$$

Signalons encore quelques formules résultant des précédentes:

$$P_1 \underset{i}{\llcorner} P_2 \overset{k}{\underset{j}{\sqsubset}} = P_1 \wedge P_2 \overset{k}{\ulcorner} P_1 \wedge \bar{P}_2 \overset{j}{\underset{i}{\sqsubset}},$$

$$P_1 \overset{i}{\ulcorner} P_2 \overset{k}{\underset{j}{\sqsubset}} = P_1 \vee \bar{P}_2 \underset{k}{\llcorner} P_1 \vee P_2 \overset{i}{\underset{j}{\sqsubset}},$$

$$P_1 \overset{i}{\ulcorner} P_2 \overset{l}{\underset{k}{\sqsubset}} = P_2 \underset{m}{\llcorner} P_1 \overset{i}{\underset{l}{\sqsubset}} \underset{m}{\lrcorner} P_1 \overset{i}{\underset{k}{\sqsubset}},$$

$$P_1 \underset{i}{\llcorner} P_2 \overset{l}{\underset{k}{\sqsubset}} = P_2 \underset{m}{\llcorner} P_1 \overset{l}{\underset{i}{\sqsubset}} \underset{m}{\lrcorner} P_1 \overset{k}{\underset{i}{\sqsubset}}.$$

Si $P_1 = P_1 \wedge P_2$, alors

$$P_1 \overset{i}{\ulcorner} P_2 \overset{l}{\underset{k}{\sqsubset}} = P_2 \underset{k}{\llcorner} P_1 \overset{i}{\underset{l}{\sqsubset}}.$$

Si $P_1 \wedge P_2 = 0$, alors

$$P_1 \overset{i}{\ulcorner} P_2 \overset{l}{\sqsubset}_k = P_2 \overset{l}{\ulcorner} P_1 \overset{i}{\sqsubset}_k .$$

§ 4.9. Transformations équivalentes des opérateurs non logiques

Si, pour tout état I, aux valeurs des opérateurs d'action D_1 et D_2 correspondent les complexes équivalents dont les cortèges d'entrée et de sortie sont respectivement égaux, alors on a

$$D_1 \underset{i}{\llcorner} = D_2 \underset{i}{\llcorner} .$$

Dans les mêmes conditions, sans tout de même exiger que les cortèges de sortie des complexes se composent des mêmes cellules, on a

$$D_1 T = D_2 T .$$

Si à deux opérateurs de variation V_1 et V_2 correspondent les complexes équivalents dont les cortèges d'entrée et de sortie sont respectivement égaux, on a

$$V_1 \underset{i}{\llcorner} = V_2 \underset{i}{\llcorner} .$$

Si, pour tout état I, le complexe des valeurs d'un opérateur d'action D représente le produit des complexes des valeurs de deux opérateurs D_1 et D_2, alors

$$D \underset{i}{\llcorner} = D_1 D_2 \underset{i}{\llcorner} .$$

Si le complexe d'un opérateur de variation V représente le produit des complexes de deux opérateurs de variation V_1 et V_2, alors

$$V \underset{i}{\llcorner} = V_1 V_2 \underset{i}{\llcorner} .$$

Les deux dernières formules permettent de décomposer un opérateur non logique en deux, ou inversement (si l'on permute les membres dans chaque formule), de réunir deux opérateurs non logiques successifs en un seul.

Si les cortèges d'entrée et de sortie d'un opérateur non logique D ou V, pour tout état I, sont disjoints et que leurs cortèges intermédiaires n'ont pas de cellules communes avec ceux d'entrée, alors

$$D \underset{i}{\llcorner} = DD \underset{i}{\llcorner} ,$$

$$V \underset{i}{\llcorner} = VV \underset{i}{\llcorner} .$$

Si, pour tout état I, le complexe des valeurs d'un opérateur D_2 est inverse à droite au complexe des valeurs d'un opérateur D_1, alors

$$\sum \{\pi D_1 D_2 \underset{i}{\llcorner} ,\ \underset{i}{\lrcorner}\} = \sum \{\Lambda,\ \pi \underset{i}{\lrcorner}\}$$

ou, dans le cas spécial,

$$\pi D_1 D_2 \underset{i}{\llcorner} \underset{i}{\lrcorner} = \pi \underset{i}{\lrcorner}.$$

Si le complexe des valeurs d'un opérateur V_2 est inverse à droite de celui d'un opérateur V_1, alors

$$\sum \{\pi V_1 V_2 \underset{i}{\llcorner}, \underset{i}{\lrcorner}\} = \sum \{\Lambda, \pi \underset{i}{\lrcorner}\}$$

ou, dans le cas spécial,

$$\pi V_1 V_2 \underset{i}{\llcorner} \underset{i}{\lrcorner} = \pi \underset{i}{\lrcorner}.$$

Si deux schémas $\sum \{\pi V \underset{i}{\llcorner}, \underset{i}{\lrcorner}\}$ et $\sum \{\Lambda, \pi \underset{i}{\lrcorner}\}$ ne contiennent aucun opérateur qui dépende essentiellement des paramètres appartenant au cortège de sortie de l'opérateur V, alors

$$\sum \{\pi V \underset{i}{\llcorner}, \underset{i}{\lrcorner}\} = \sum \{\Lambda, \pi \underset{i}{\lrcorner}\},$$

ou, en particulier,

$$\pi V \underset{i}{\llcorner} \underset{i}{\lrcorner} = \pi \underset{i}{\lrcorner}.$$

Soient $i_1, i_2, \ldots, i_t$ les paramètres d'un schéma $\sum$. On peut introduire un nouveau paramètre α vérifiant la relation

$$\alpha \equiv \varphi (i_1, i_2, \ldots, i_t)$$

et remplacer, partout ou dans certains endroits, dans les relations correspondant aux opérateurs logiques et dans les indices des opérateurs dépendant des paramètres, l'expression $\varphi (i_1, i_2, \ldots, i_t)$ par la lettre α. De plus, il faut écrire à la fin de la suite normale de chaque opérateur de variation la formule

$$\alpha := \varphi (i_1, t_2, \ldots, i_t)$$

et inclure α dans le cortège I.

§ 4.10. Permutations des opérateurs

Si, pour tout état I (collection des valeurs des paramètres) les cellules du cortège de sortie d'un opérateur d'action D_i $(i = 1, 2)$ sont des arguments non essentiels du complexe de l'opérateur D_j $(j=1, 2; j \neq i)$, et que le cortège intermédiaire de chaque opérateur n'a pas de partie commune avec le cortège d'entrée ni avec le cortège de sortie de l'autre, alors on a

$$D_1 D_2 \underset{k}{\llcorner} = D_2 D_1 \underset{k}{\llcorner}.$$

Dans les conditions analogues, et dans la supposition supplémentaire que les deux opérateurs ne dépendent pas de paramètres, on a la relation

$$V_1 V_2 \underset{k}{\llcorner} = V_2 V_1 \underset{k}{\llcorner}.$$

Si les cellules du cortège de sortie d'un opérateur V ne sont pas des paramètres essentiels d'un opérateur d'action D, alors

$$VD \llcorner_{i} = DV \llcorner_{i}.$$

Soient K le complexe d'un opérateur d'action D pour un état quelconque du schéma, X et Y, ses cortèges d'entrée et de sortie. Soit, de plus $\Psi(Y, Z)$ la relation (pour le même état du schéma) contrôlée par un opérateur logique P, les Y et Z étant disjoints. Alors on a

$$DP \underset{i}{\overset{j}{\sqsubset}} = P' \overset{m}{\ulcorner} D \llcorner_{i} \lrcorner_{m} D \llcorner_{j},$$

où P' est un opérateur logique contrôlant la relation $\Psi[K(X), Z]$ que l'on obtient de $\Psi(Y, Z)$ en remplaçant les cellules qui appartiennent à Y par leurs expressions en fonction des cellules du cortège X au moyen de l'algorithme de substitutions décrit au § 4.2.

Si, quel que soit l'état I, les cellules du cortège de sortie d'un opérateur d'action D ne sont pas des arguments essentiels du prédicat d'un opérateur logique P, alors

$$DP \underset{i}{\overset{j}{\sqsubset}} = P \overset{m}{\ulcorner} D \llcorner_{i} \lrcorner_{m} D \llcorner_{j}.$$

Soient $\varphi_1 = \varphi_1(I)$, $\varphi_2 = \varphi_2(I)$, . . ., $\varphi_k = \varphi_k(I)$ les indices supérieurs d'un opérateur d'action qui dépend de paramètres, et V un opérateur de variation qui a K pour complexe. Désignons

$$\psi_1 = \varphi_1[K(I)], \ \psi_2 = \varphi_2[K(I)], \ldots, \psi_k = \varphi_k[K(I)].$$

Alors on a

$$VD^{\varphi_1, \varphi_2, \ldots, \varphi_k} \llcorner_{i} = D^{\psi_1, \psi_2, \ldots, \psi_k} V \llcorner_{i}.$$

Pour les mêmes désignations des indices et de l'opérateur de variation, et pour n'importe quel état du système, si le contenu de l'opérateur logique est $\Psi(I, Z)$, où les cortèges I et Z sont disjoints, alors on a

$$VP^{\varphi_1, \varphi_2, \ldots, \varphi_k} \underset{i}{\overset{j}{\sqsubset}} P'^{\psi_1, \psi_2, \ldots, \psi_k} \overset{m}{\ulcorner} V \llcorner_{i} \lrcorner_{m} V \llcorner_{j}.$$

Ici $P'^{\psi_1, \psi_2, \ldots, \psi_k}$ est un opérateur logique avec le contenu $\Psi[K(I), Z]$.

§ 4.11. Subordination d'un opérateur à un prédicat

Désignons par la lettre w une expression élémentaire contenant un opérateur Q différent de I_0 (Q peut être logique ou non).

Considérons un schéma $\sum\{w\}$ dont les indices des signes de passage diffèrent d'un nombre naturel α. Désignons par $\sum_{\alpha}\{w\}$ le

schéma qu'on obtient de $\sum\{w\}$ en écrivant $\underset{\alpha}{\rfloor}$ après son expression élémentaire quelconque. Il est évident que

$$\sum\{w\} = \sum_\alpha\{w\}.$$

S'il existe un prédicat P_0 tel que, pour toute position du signe de passage fermant $\underset{\alpha}{\rfloor}$, soit vraie la formule

$$\sum\{w\} = \sum_\alpha\{P_0 \underset{\alpha}{\llcorner} w\},$$

alors on dit que l'expression élémentaire w et l'opérateur Q sont *subordonnés* au prédicat P_0 et l'on écrit

$$w < P_0;\ Q < P_0$$

En géneral, le contenu du prédicat P_0 peut avoir des paramètres du schéma parmi ses arguments et modifier sa forme en fonction de l'état du cortège I.

Si dans le schéma $\sum\{W\}$

$$w < P_0,$$

alors on a toujours $P_0 = 1$ immédiatement avant l'exécution de w.

Pour qu'un opérateur Q du schéma $\sum\{w\}$ ne s'exécute pour aucun X appartenant à G, il faut et il suffit qu'il soit subordonné à zéro, i.e. au prédicat identiquement nul.

Si $Q < 0$, alors $Q < P_0$, où P_0 est un prédicat quelconque admissible pour le schéma.

Si $Q < P_0$, $Q < P^*$, alors

$$Q < P_0 \wedge P^*.$$

Si $Q < P_0$, alors, pour tout prédicat P^* admissible, on a

$$Q < P_0 \vee P^*$$

Si un opérateur logique P satisfait à la condition $P < P_0$, on a les formules

$$P \overset{j}{\underset{i}{\sqsubset}} = P_0 \wedge P \overset{j}{\underset{i}{\sqsubset}},$$

$$P \overset{j}{\underset{i}{\llcorner}} = \overline{P}_0 \vee P \overset{j}{\underset{i}{\sqsubset}}.$$

Pour le schéma $\sum\{P \underset{i}{\llcorner} w\}$ on a $w < P$.

Si l'on a $P < P_0$ dans le schéma $\sum\{P\lambda_i w\}$, alors $w < P_0$.

Si, pour tout état d'un schéma $\sum\{w\}$, l'opérateur Q est subordonné au prédicat P_0 de contenu $\alpha_j \equiv \psi(\alpha_1, \alpha_2, \ldots, \alpha_n)$, alors

$$w = Q_1 w.$$

où Q_1 est un opérateur non logique auquel correspond le complexe dont la suite normale se compose d'une seule formule

$$\alpha_j := \psi(\alpha_1, \alpha_2, \ldots, \alpha_n).$$

S'il n'y a pas de paramètres parmi les lettres $\alpha_1, \alpha_2, \ldots, \alpha_n$, alors Q_1 est un opérateur d'action; si α_j et tous les arguments essentiels de ψ sont des paramètres, alors Q_1 est un opérateur de variation. Nous ne considérons pas ici les autres cas.

Si $Q < P_0$ dans $\sum\{w\}$, et que P_0 ait le contenu

$$i_j = \theta(i_1, i_2, \ldots, i_t),$$

alors (dans la notation d'indices introduite plus haut) les indices $\varphi_1, \varphi_2, \ldots, \varphi_k$ de l'opérateur Q (tous ou quelques-uns) peuvent être remplacés par les indices

$$\psi_1 = \varphi_1[K(I)],\ \psi_2 = \varphi_2[K(I)],\ \ldots,\ \psi_k = \varphi_k[K(I)].$$

Ici K est un complexe dont la suite normale se réduit à

$$i_j = \theta(i_1, i_2, \ldots, i_t).$$

§ 4.12. Complétude du système des transformations équivalentes des algorithmes. Leur domaine d'application

Le système décrit des transformations des algorithmes est complet pour les complexes uniformes, ainsi que pour les algorithmes droits ou se réduisant aux algorithmes droits, au problème de l'identité près (i.e. sous la condition que, chaque fois qu'on a besoin d'utiliser une identité d'opérations, on possède une information permettant de reconnaître l'identité). Le problème de la complétude du système des transformations pour les algorithmes arbitraires n'est pas étudié. Le théorème connu de Gödel fait supposer que le système n'est pas complet, comme d'ailleurs toute théorie assez compliquée.

Les transformations équivalentes, effectuées formellement ou intuitivement, sont largement employées dans la programmation. Or, leur domaine d'application dépasse de loin la programmation. Une algorithmisation efficace de divers processus est impensable sans transformations équivalentes des algorithmes. Une grande importance et un grand intérêt reviennent au problème des transformations équivalentes orientées. On peut les utiliser dans les compilateurs si l'on veut améliorer automatiquement les programmes au cours de l'assemblage (voir le p. 5.2.2). Leur utilisation dans les interprétateurs permet de perfectionner les algorithmes dynamiquement, au cours de leur exécution (voir le p. 5.2.2.1).

CHAPITRE 5

SOFTWARE

§ 5.1. Notion de software

L'idée de munir un calculateur d'un système développé de programmes fut née des besoins pratiques. La programmation exigeant beaucoup de travail et de dépenses, on a intérêt de conserver les programmes en vue de leur utilisation multiple. Or, aucune collection de programmes ne peut satisfaire à tous les besoins, d'où la nécessité d'en composer d'autres. Pour faciliter les travaux de programmation et les rendre moins coûteux, on confie au calculateur la réalisation de certaines étapes. Cela exige évidemment la création de programmes auxiliaires, destinés non pas à résoudre tel ou tel problème, mais à aider à la programmation. Ce n'est là qu'un moyen d'augmenter l'efficacité du calculateur. Une analyse du fonctionnement du calculateur montre qu'une grande partie de ses équipements n'est pas utilisée au cours d'exécution d'un programme. Dans les ensembles électroniques modernes de nombreuses simultanéités sont possibles, plusieurs problèmes peuvent être traités à la fois. Pendant que les dispositifs de mémoire s'échangent de l'information concernant l'un des problèmes, l'unité de calcul exécute le programme d'un autre, le dispositif de sortie fournit les résultats d'un troisième, le dispositif d'entrée introduit les données d'un quatrième, etc. Pour assurer un tel régime de fonctionnement de la machine, il faut, premièrement, choisir convenablement la collection des problèmes à résoudre simultanément et, deuxièmement, distribuer le travail entre les dispositifs, au cours de la résolution, d'une manière rapide et correcte.

On ne peut confier cette tâche à l'opérateur par suite de son extrême complicité; le niveau technique actuel ne permet non plus de le faire en munissant la machine de dispositifs supplémentaires. Il reste une seule possibilité: de commander l'exécution des programmes et le fonctionnement de divers organes du calculateur au moyen d'un programme spécial. Un tel programme moniteur doit se trouver constamment dans la mémoire du calculateur, et chacun des programmes en cours doit s'y adresser au moment opportun pour qu'il décide comment distribuer le travail à poursuivre parmi les organes du calculateur. Le régime décrit s'appelle travail en multiprogrammation.

L'efficacité de l'utilisation des calculateurs étant ainsi élevée, l'emploi du programme moniteur suggéra une nouvelle idée : *d'élargir et de modifier au moyen d'un programme* les fonctions de l'unité de commande du calculateur. En munissant la machine d'un moniteur de gestion dûment composé, on peut la transformer, du point de vue du programme, en une machine tout à fait différente. La nouvelle conception du calculateur, basée sur cette idée, a permis d'utiliser une même machine de plusieurs manières, selon le programme moniteur : soit pour le travail en multiprogrammation, soit en temps réel (en tant que machine de gestion), soit en temps partagé, lorsque le temps du calculateur est vraiment partagé entre plusieurs utilisateurs et cela d'une telle façon que chacun a l'impression de bénéficier de toute la machine, etc.

C'est ainsi qu'est née l'idée d'équiper le calculateur d'un grand nombre de programmes qui : a) permettent de résoudre certains problèmes sans programmation ; b) assument certaines tâches de programmation ; c) choisissent le régime de fonctionnement avantageux du calculateur.

On appela l'ensemble de tels programmes *système d'exploitation du calculateur* ou *système* tout court *). Il est clair que chaque programme composé pour un calculateur donné n'appartient pas à son système (il y a des programmes qu'on compose et qu'on met en œuvre à l'aide des programmes du système sans les y inclure). C'est ainsi que se forma la notion intuitive de software. L'étape suivante consista en la recherche d'une définition formelle de la nouvelle notion.

On accepta avant tout que les éléments du système étaient des programmes et non pas les méthodes se trouvant à la base de programmes, ni les algorithmes composés dans des langages différents de celui de machine, ni enfin les langages dans lesquels on formule ces méthodes, algorithmes ou programmes. C'est-à-dire, on établit que le système du calculateur était un ensemble de programmes. Certains auteurs commencèrent même d'employer le terme « système de programmes » au lieu de « système d'exploitation » (v. [60]). D'autres pensèrent qu'il suffisait d'ajouter que les programmes en question étaient créés pour rendre plus aisée l'exploitation du calculateur par les utilisateurs. L'insuffisance d'une telle « définition » saute aux yeux, car elle permet d'inclure dans le système n'importe quel programme, du moment que son auteur déclare qu'il l'a composé dans le but ci-indiqué. La mention que les programmes doivent s'utiliser régulièrement pour l'exploitation du calculateur ne sauve pas l'affaire. En effet, un programme peut être utilisé régulièrement

*) Pour désigner cet ensemble de programmes comprenant les aides à la programmation et les aides à l'exploitation on emploie souvent le terme anglo-saxon *software*.

dans un centre de calcul et ne pas l'être du tout dans un autre. Par conséquent, il appartient au système du point de vue des uns et n'y appartient pas selon d'autres.

Pour remédier à cet inconvénient, certains auteurs (voir, par exemple, [60]) proposent d'énumérer les programmes appartenant au système et de définir ainsi cette notion. Pourtant, ils sont obligés d'admettre que le système n'est pas invariable, donc les noms des programmes indiqués dans la définition sont des mots n'ayant pas de valeur exacte, puisque les objets qu'ils désignent changent, on ne sait pas dans quelle mesure.

Remarquons encore que l'énumération permet de définir un système concret, tandis qu'on a besoin d'une définition exacte de cette notion pour confronter différents systèmes.

Il paraît qu'on peut définir intuitivement le système d'un calculateur comme un ensemble de programmes dont chacun *peut être employé en pratique* par un utilisateur, seul ou en combinaison avec d'autres programmes (selon leur destination), soit pour résoudre les problèmes, soit pour exécuter certains travaux liés à la programmation, soit pour créer un régime déterminé de fonctionnement du calculateur. Une telle définition peut être formalisée de la manière suivante.

Soit K un ensemble de calculateurs. Désignons par F un système des règles déterminant la structure des programmes, et par Φ un système des règles d'après lesquelles on présente la documentation accompagnant chaque programme ou groupe de programmes. Soit B une bibliothèque de programmes. Nous dirons qu'un programme appartient au système de l'ensemble K s'il satisfait aux conditions suivantes:

1) il s'emploie pour l'un au moins des calculateurs de l'ensemble K;

2) il satisfait au système de règles F;

3) il satisfait au système de règles Φ;

4) il est classé dans la bibliothèque B, i.e. se trouve dans un endroit déterminé du dépôt des programmes, sous une forme convenable, et figure dans le catalogue de la bibliothèque.

En particulier, l'ensemble de calculateurs peut se réduire à une seule machine.

La condition 1) ne demande pas d'explications, sa nécessité est évidente. La condition 2) garantit une utilisation conjointe des programmes. Si ce n'est pas le cas, F peut être vide. Alors le système ne contiendra aucun programme déterminant le mode d'utilisation du calculateur. Des systèmes de ce type se rencontraient à l'époque des calculateurs de la 1[re] génération. A présent, on les trouverait primitifs.

La condition 3) assure la possibilité d'utiliser chaque programme, d'apprendre tout ce qu'il faut pour son application. Outre la des-

cription détaillée, chaque programme est accompagné d'une série d'indications concernant son mode d'emploi, la préparation de l'information, etc.

La condition 4) permet de se rendre compte de l'existence du programme dans le système et de le trouver s'il le faut (à la main ou automatiquement, selon la bibliothèque). Remarquons que, si la recherche des programmes dans la bibliothèque se fait par la machine même, au moyen d'un programme spécial, ce programme doit lui-même appartenir au système, donc se trouver dans la bibliothèque.

Définitivement, on peut dire que le système d'exploitation d'un ensemble de calculateurs (en particulier, d'un calculateur unique) représente un système de programmes rangés dans une bibliothèque B et satisfaisant aux deux systèmes de règles suivants: F, qui détermine la structure des programmes et Φ, qui établit les indications d'accompagnement pour chaque programme ou groupe de programmes.

§ 5.2. Classification des programmes du software

Notre définition donne la syntaxe du software. Sa sémantique peut être donnée, dans les cas concrets, en énumérant les programmes ou les systèmes de programmes. On peut également dire que la sémantique d'un système est contenue dans les documents accompagnant ses programmes.

Dans la plupart des cas, le software de calculateurs comprend *) les groupes de programmes suivants:

1) système d'exploitation;
2) système des moyens de programmation;
3) annexes;
4) système de programmes destinés à maintenir le software à l'état de service.

On doit y ajouter encore le

5) système de programmes tests destinés à contrôler l'état du calculateur.

Cette partie du système est dans la plupart des cas ignorée par les programmeurs, n'étant utilisée que par les techniciens desservant les calculateurs.

5.2.1. Système d'exploitation. Le *système d'exploitation* contient les programmes qui déterminent le mode d'exploitation du calculateur et élargissent ses possibilités opérationnelles. Du point de vue du programmeur, le système d'exploitation est inséparable

*) D'après la classification admise actuellement (comparer avec la classification du § 5.3).

du calculateur, ils forment ensemble ce qu'on appelle *système d'exécution*. Ce dernier représente un nouveau calculateur, différent de l'ensemble électronique de base.

Le système d'exploitation se compose d'une série de programmes dont les principaux sont *):

— le *dispatcher*, un programme qui réalise la « stratégie » du fonctionnement du calculateur, prévue par le système d'exploitation, i.e. un mode d'exploitation détérminé;

— le *superviseur* ou le *moniteur*, un programme qui réalise la « tactique » du travail, proposée à la machine par l'opérateur humain, en mode d'exploitation établi;

— des programmes de service, tels que les *programmes d'entrée* des données, les *programmes d'édition* et *de sortie* des résultats, le programme *chargeur* qui met en mémoire rapide les programmes d'utilisateurs, le *bibliothécaire*, i.e. le programme d'appel des sous-programmes pour l'exécution des *macro-instructions*, les *sous-programmes de macro-instructions* eux-mêmes, le *programme de conversation* du système d'exploitation avec l'opérateur humain, etc.

Les systèmes d'exploitation des calculateurs modernes bénéficient de quatre moyens puissants et souples (que les machines de la 1[re] génération ne possédaient pas): il s'agit du système développé d'interruptions, de la protection de la mémoire, de la protection des instructions et d'emploi de masques d'interruptions (voir p. 1.1.4).

5.2.1.1. Dispatcher. Le dispatcher se distingue par le fait que c'est lui seul qui reçoit la commande après une interruption dans l'exécution de n'importe quel autre programme. Etant lui-même protégé contre les interruptions, il examine périodiquement le registre d'interruptions et « apprend » les interruptions qui ont eu lieu pendant son fonctionnement. La zone de mémoire rapide contenant le dispatcher est protégée contre toute écriture parasite par un programme extérieur.

L'essentiel du fonctionnement du dispatcher consiste en ce qu'il classe toutes les interruptions en tactiques et en stratégiques. Dans le premier cas il passe immédiatement la commande et l'information nécessaire au superviseur; dans le second, il décide lui-même de la réaction convenable sur l'interruption. Appelons conclusion le texte qui correspond à cette réaction. Après avoir formulé une conclusion, le dispatcher l'inscrit sur la liste des conclusions, effectue un nouvel examen du registre d'interruptions et, s'il n'y en a plus, passe la commande en se conformant: a) à la liste des conclu-

*) Il n'existe pas à l'U.R.S.S., ni à l'étranger, de terminologie arrêtée, ni même de « répartition du travail » communément admise entre les programmes du système d'exploitation. Nous utilisons la terminologie et la répartition des tâches qui sont, à notre avis, les plus adéquates.

sions où il choisit une conclusion par ordre d'inscription ; b) à la disponibilité des organes du calculateur.

La commande peut être passée au dispatcher non seulement par le dispositif d'interruptions, mais aussi par d'autres programmes. Alors ses actions sont toujours conditionnées par la stratégie qu'il réalise, de même que dans le cas précédent.

Exemple 5.1. En exécutant un paquet de programmes, un dispatcher qui assure un travail en multiprogrammation peut fonctionner de la manière suivante. (Nous appellerons programme d'utilisateur n'importe quel programme autre que dispatcher ou superviseur.)

1°. La commande est reçue à la suite d'une interruption. Passage au p. 4°.

2°. La commande est transférée par le superviseur. Passage au p. 8°.

3°. La commande est reçue à partir d'un programme d'utilisateur. Passage au p. 10°.

4°. Vérification si l'interruption est tactique. Si oui, formation d'un message contenant sa description, passage de la commande au superviseur ; si non, passage au p. 5°.

5°. Formation de la conclusion sur l'interruption (stratégique) en question, son inscription sur la liste des conclusions. Passage au p. 6°.

6°. Examen du registre d'interruptions. S'il n'y en a pas, passage au p. 7°, autrement au p. 4°.

7°. Passage de la commande au programme d'utilisateur ou au superviseur selon la liste des conclusions et la stratégie du travail.

8°. Vérification s'il y a un message du superviseur. S'il n'y en a pas, passage au p. 6°, autrement au p. 9°.

9°. Correction de la liste des conclusions d'après le message du superviseur. Passage au p. 6°.

10°. Si le programme d'utilisateur est exécuté définitivement, passage au p. 6°, sinon au p. 11°.

11°. Si le programme d'utilisateur est exécuté mais il faut encore sortir les résultats, passage au p. 12°, sinon au p. 13°.

12°. Inscription de la conclusion sur le transfert de la commande au programme d'édition et de sortie des résultats sur la liste des conclusions. Passage au p. 6°.

13°. Si le programme d'utilisateur est exécuté mais il faut encore exécuter un programme supplémentaire (se trouvant dans la mémoire rapide), passage au p. 14°, sinon au p. 15°.

14°. Formation et inscription sur la liste de la conclusion sur le transfert de la commande à ce programme supplémentaire. Passage au p. 6°.

15°. Si le programme d'utilisateur demande l'entrée d'une information, passage au p. 16°, sinon au p. 17°.

16°. Formation de la conclusion sur l'introduction de l'information. Passage au p. 6°.

17°. Si le programme d'utilisateur demande l'exécution d'une macro-instruction, formation de la conclusion sur le transfert de la commande au bibliothécaire. Passage au p. 6°.

Dans le processus décrit, la stratégie du calculateur est concentrée surtout au p. 7°. En particulier, le fonctionnement d'un dispatcher en monoprogrammation sera, en grands traits, le même. Seul le contenu du p. 7° sera différent.

5.2.1.2. **Superviseur.** Le superviseur, après s'être acquitté de ses fonctions, passe toujours la commande au dispatcher et ne peut la reprendre par la suite que du dispatcher. Or, pendant son fonctionnement, il peut appeler les sous-programmes. Il établit sur demande de l'opérateur humain l'ordre d'exécution des programmes et distribue entre eux l'équipement disponible du calculateur. Les actions du superviseur sont conformes aux ordres transmis à partir du pupitre de commande par l'opérateur avec lequel le superviseur est en communication. Il l'informe que ses ordres sont compris, qu'ils s'exécutent de telle ou telle manière, qu'une situation demande son intervention. Un sous-programme spécial de « dialogue » est prévu à cette fin dans le superviseur. Le superviseur est protégé contre les autres programmes, mais non contre les interruptions *). Pourtant, en cas d'interruption pendant son fonctionnement, le dispatcher lui remet la commande dès que l'interruption est enregistrée. Lorsque l'exécution des programmes mis sous la surveillance du dispatcher par le superviseur se termine, celui-ci appelle (en s'adressant pour cela au chargeur) le programme suivant du paquet, et le met sous les ordres du dispatcher. Ainsi, le superviseur dirige continûment l'exécution des programmes en mode d'exploitation réalisé par le dispatcher. En résumant, on peut dire que le superviseur résout deux problèmes principaux : 1) la gestion du travail du calculateur ; 2) la communication avec l'opérateur humain. Le caractère du fonctionnement du superviseur dépend du mode d'exploitation adopté. Par exemple, en fonctionnement en multi-consoles, le superviseur communique avec tous les opérateurs, en surveillant la marche de tous les travaux accomplis sous leur direction.

5.2.1.3. **Travaux accomplis par le système exécutif.** Pour le système exécutif, les programmes du software qui n'appartiennent pas au système d'exploitation sont au même niveau que les programmes d'utilisateurs. Il s'agit des programmes de programmation automatique, de mise au point des programmes rédigés, etc.

---

*) En exploitation en multiprocesseur, le superviseur est protégé contre toutes les interruptions qui ne proviennent pas du pupitre de commande.

C'est pour cette raison que, lorsqu'il s'agit du fonctionnement du système exécutif, on remplace la notion de « résolution d'un problème » par celle d'« exécution d'un travail » que les utilisateurs de la machine considèrent comme plus large *).

Citons quelques exemples de travaux:

— traduction d'un langage algorithmique dans le langage du système exécutif;

— mise au point d'un programme au moyen du calculateur;

— préparation de l'information pour la résolution d'un problème;

— résolution d'un problème à l'aide d'un programme déjà prêt;

— édition et impression des résultats de résolution d'un problème;

— enfin, il y a des travaux plus compliqués, représentant des combinaisons des travaux énumérés simples.

Il découle de ce qu'il a été dit que la notion de travail comprend l'exécution des programmes d'entrée et de sortie, lorsqu'elle représente un problème indépendant.

5.2.1.4. Entrée et sortie de l'information. Notion de fichier. L'information rangée dans la mémoire rapide peut être considérée comme un ensemble de mots repérés par une adresse particulière (voir l'exemple 2.29). Quant à l'information d'entrée, la situation est toute différente. Elle se présente comme une suite de symboles, portés sous forme de perforations sur un support (ruban ou cartes). L'information de sortie a également la forme d'une suite de lignes.

Le plus souvent, l'information perforée dans un support mécanographique est transférée dans une mémoire extérieure où elle reste jusqu'à ce qu'elle ne soit nécessaire à la résolution d'un problème, alors elle est introduite dans la mémoire rapide. La lecture peut se faire par parties.

Les programmes d'entrée de l'information dans la mémoire extérieure et les programmes de transfert de l'information de la mémoire extérieure dans la mémoire rapide forment le système des programmes d'entrée.

Les calculateurs modernes traitent souvent des problèmes dont les données initiales sont très volumineuses. Pour éviter les erreurs lors des transferts d'information, il faut prendre certaines précautions. Les fautes que peuvent commettre les opérateurs ou les programmeurs peu expérimentés sont prévenues par des mesures spé-

*) Il va de soi qu'en réalité l'exécution d'un travail par le système exécutif n'est rien d'autre que la résolution d'un problème. Par conséquent, il ne s'agit pas d'une généralisation ou d'un élargissement de cette notion. C'est plutôt une conception plus étroite du terme « problème » qu'on admet.

ciales qui consistent en ce que l'information est munie d'indices supplémentaires.

On convient que chaque dispositif physique destiné à garder l'information peut contenir un *volume d'information* (on considère comme tel dispositif soit une bobine de bande magnétique, soit un tambour magnétique, soit un paquet de disques magnétiques). En début de chaque volume on met son *étiquette de tête*, et à la fin, l'*étiquette de fin de volume.*

Ainsi un volume représente tout le texte qui commence par une étiquette de tête, se termine par une étiquette de fin de volume et qui est enregistré dans un dispositif indépendant. Les volumes sont numérotés.

Introduisons la notion de fichier.

On appelle fichier une suite d'éléments d'information qui sont liés entre eux (appartenant à un même problème par exemple). Un fichier est entré ou sorti de la machine comme un bloc unique d'information. Il peut occuper un ou plusieurs volumes consécutifs, ou bien une partie de volume.

La structure d'un fichier dépend des particularités du système d'exploitation, en particulier, des programmes d'entrée et de sortie. On dispose en début du fichier une information de service qui indique son utilisateur et son appartenance à un problème déterminé. De plus, on donne des précisions sur la structure du fichier (un fichier peut être composé de blocs d'information, des données isolées ou des groupes de données repérés d'une manière déterminée, etc.), sur la nature de l'information qu'il contient (méthode de codage), etc.

L'information auxiliaire dépend du type de mémoire réservée à la conservation du fichier. Si un fichier est enregistré sur bande magnétique, il est à l'accès séquentiel. On en tient compte en choisissant la place de l'information auxiliaire, de façon à éviter des rebobinages inutiles. Si un fichier est constitué sur disques, on a l'accès direct à ses éléments. On en tient compte dans sa structure et dans l'organisation de l'information auxiliaire.

Sans nous arrêter sur les détails d'ordre technique (on peut les trouver dans [12]), remarquons qu'à l'intérieur d'un fichier l'information peut être divisée en *blocs* qui, à leur tour, se composent d'*enregistrements.*

S'il faut sortir une information du calculateur on doit l'avoir sous forme d'un fichier.

Les systèmes d'exploitation existants ne détruisent généralement pas un fichier au cours du travail pour lequel il a été créé. L'édition de l'information se fait dans le cadre d'un fichier: on la divise en parties, on y introduit des symboles d'espace, des séparateurs nécessaires, etc. Ceci fait, on effectue la sortie et l'impression de l'information. On a déjà dit que ces opérations sont réalisées par les programmes de sortie faisant partie du système d'exploitation.

5.2.1.5. Modes d'exploitation du calculateur. Banque des données. Nous avons mentionné les différents modes de fonctionnement du calculateur dont la réalisation est l'une des préoccupations principales du dispatcher. Voyons-les de plus près.

Certains modes d'exploitation sont prévus pour la résolution des problèmes que l'on met sous la forme d'un *train de programmes.* Chaque train de programmes est pourvu de précisions sur les programmes à exécuter et sur leurs priorités respectives (système de *priorités*).

L'exécution d'un train de programmes se fait sous la surveillance du superviseur. La machine peut être alors exploitée en mono- ou multiprogrammation. L'efficacité de l'utilisation du calculateur dépend fortement de la manière dont les travaux sont groupés. Un train est considéré comme bien composé, lorsque le processeur central (l'unité de calcul et l'unité de commande) ne chôme pas. Le désir de bien composer un certain train peut avoir pour conséquence que les autres groupes seront très mal composés. Il faut donc veiller à ce que les trains de programmes soient également bons.

Il est utile de donner la priorité inférieure au programme dont l'exécution occupe presqu'exclusivement le processeur central. Le système exécutif en chargera le processeur central chaque fois qu'il est libre. Un tel travail est habituellement dit *de fond.*

Les modes d'exploitation décrits sont appelés *traitement en multiprogrammation.* Un calculateur moderne peut traiter à la fois jusqu'à 16 programmes.

Les possibilités du système exécutif dépendent dans une grande mesure des moyens techniques du calculateur. L'insuffisance de l'équipement rend impossible le travail en multiprogrammation.

De plus, les systèmes d'exploitation dépendent fortement des problèmes auxquels le calculateur est surtout destiné. Actuellement, on considère comme les plus importants les problèmes scientifiques, techniques et économiques. C'est vers ces problèmes que sont orientés les systèmes d'exploitation les plus connus. En multiprogrammation, le temps d'exécution de chaque programme isolé croît, en revanche le temps moyen de leur exécution se trouve réduit. Le temps que met la machine pour exécuter les programmes d'un train est moindre que celui qu'elle mettrait pour exécuter ces programmes l'un après l'autre.

En plus des modes d'exploitation en multiprogrammation il faut signaler l'utilisation de la machine en *temps partagé.* Ce régime est caractérisé par le fait qu'un grand nombre de dispositifs d'entrée-sortie à distance, que l'on appelle *terminaux*, sont connectés au calculateur. Il est même possible que les terminaux se trouvent dans des villes différentes.

Dans le mode d'exploitation en multiprogrammation les utilisateurs ne sont pas admis au pupitre de commande du calculateur,

alors que dans le « temps partagé », chaque utilisateur converse directement avec la machine. Le dispatcher interroge les terminaux dans un ordre établi une fois pour toutes. Si l'un des terminaux envoie le signal de commencement du travail de l'utilisateur correspondant, le dispatcher passe la commande au superviseur qui inscrit le travail donné sur la liste des travaux à exécuter en simultanéité et repasse la commande au dispatcher. Celui-ci appelle à tour de rôle les programmes d'utilisateurs pour exécution pendant une tranche de temps relativement courte. Au fur et à mesure d'apparition des résultats, ils sont envoyés dans le canal de l'utilisateur intéressé. Par intervalles de quelques secondes, le système sert peu à peu chaque utilisateur. Celui-ci a l'impression de disposer personnellement de la machine.

L'exploitation en temps partagé ne poursuit pas le but d'obtenir un rendement maximal de l'équipement du calculateur, mais d'améliorer la communication entre la machine et les utilisateurs.

Ce mode d'utilisation est commode dans les cas où le travail doit se poursuivre sous forme de conversation entre le calculateur et l'utilisateur. Ceci a lieu, par exemple, au cours de la mise au point d'un programme sur le calculateur, dans le cas des problèmes de l'informatique du type question — réponse. Si le nombre de terminaux est grand, ce mode d'exploitation est avantageux pour l'entrée d'une grande quantité d'information, car on n'a pas besoin de perforer l'information et, de plus, il est possible d'effectuer, parallèlement, non seulement le contrôle d'entrée (sur l'écran de visualisation), mais aussi la correction d'erreurs.

On peut employer le régime du temps partagé, sous une forme un peu plus rigide, pour organiser le travail des machines de gestion. Dans ce cas, l'information est introduite à partir des dispositifs dits terminaux légers qui caractérisent l'état de l'objet commandé. Les résultats ont la forme des ordres aux organes de cet objet.

Le fonctionnement du système exécutif doit être organisé de façon que la réponse soit retardée par rapport au signal d'entrée d'un intervalle de temps admissible pour l'objet commandé.

On dit alors que le système exécutif (et le calculateur) travaille *en temps réel*.

Souvent l'exploitation en temps partagé est combinée avec multiprogrammation, le temps inutilisé par les interlocuteurs entre lesquels on partage les capacités de la machine étant employé à l'exécution d'un train de programmes qui constituent les programmes de fond.

Lorsqu'une machine fonctionne en temps partagé, le problème de conservation de l'information initiale et intermédiaire acquiert une importance particulière. Du moment qu'un utilisateur est admis au travail, le système doit trouver rapidement dans ses mémoires l'information concernant ses travaux. Pour faciliter la recherche,

l'information concernant chaque utilisateur doit être munie d'indications spéciales. Toute l'information s'accumulant dans les mémoires du calculateur doit être décrite dans une table qui représente son catalogue. Le software comporte toujours des programmes de gestion de l'information. Les blocs d'information se rapportant à divers utilisateurs et leurs problèmes, le catalogue et les programmes de gestion d'information et du catalogue s'appellent *banque des données.*

Par son caractère, la banque des données est analogue à la bibliothèque de programmes décrite au p. 5.2.2.3. Au fond, elle représente la bibliothèque des fichiers. Une fois le travail d'un utilisateur terminé, toute l'information concernant son problème revient à la banque des données ou, si elle ne s'y trouvait pas avant, est incluse dans la banque. Lorsqu'on organise une banque des données, on prend des mesures spéciales pour protéger l'information contre les utilisateurs qu'elle ne concerne pas. A cette fin, on se sert le plus souvent d'un système de « mots de passe ». Chaque utilisateur inclut dans son information un ou plusieurs codes. Sans mentionner ces codes il est impossible d'appeler l'information correspondante.

5.2.2. Programmes d'aide à la programmation. Parmi ces programmes figurent les traducteurs, programmes pour la traduction des algorithmes donnés dans différents langages symboliques (§ 3.2), dans le langage de machine, ainsi que les programmes de mise au point destinés à faciliter la correction des programmes au moyen du calculateur.

Un système d'aides à la programmation contient habituellement les traducteurs des langages algorithmiques de trois niveaux:

1. Traducteur de l'autocode 1 : 1 (on appelle ainsi un langage algorithmique très simple qui permet de composer des programmes qu'il est possible de rédiger directement dans le langage du système exécutif).

2. Traducteurs de certains langages algorithmiques orientés vers une classe de problèmes déterminée. Le nombre et le type de ces traducteurs est déterminé par la classe des problèmes que le calculateur est destiné à résoudre.

3. Traducteurs d'un ou de plusieurs langages universels indépendant de la machine, tels que ALGOL, FORTRAN ou PL/1.

5.2.2.1. Compilation et interprétation. En rapport avec la traduction d'un langage algorithmique en langage du système exécutif, il faut dire qu'il y a deux façons de combiner la traduction d'un algorithme et son exécution par la machine.

Selon la première méthode, que l'on a appelée *compilation*, l'algorithme est d'abord complètement traduit dans le langage machine et puis exécuté. On parle d'une « compilation » tout en

ayant en vue le procédé de réunion, d'assemblage des parties d'algorithmes préparées à l'avance (des sous-programmes), correspondant aux parties déterminées de l'algorithme à traduire. Par la suite, on a appliqué la même appellation à la traduction « dynamique », effectuée sans utiliser les textes préparés à l'avance.

La deuxième façon de combiner la traduction et l'exécution d'un algorithme fut nommée *interprétation*. Elle consiste en ce que l'exécution d'une partie (généralement, d'une instruction) de l'algorithme suit immédiatement sa traduction. Dans ce mode de traduction, un même élément de l'algorithme est traduit et exécuté autant de fois qu'il doit s'exécuter.

Une fois sa tâche terminée, le compilateur n'est plus nécessaire à l'exécution de l'algorithme et donc on n'a pas besoin de le conserver dans la mémoire rapide du calculateur. Par conséquent, la compilation peut être réalisée par un calculateur et le traitement du programme, par un autre, à savoir par le calculateur dont le langage est celui du programme objet.

L'application de la méthode d'interprétation exige la présence du programme interprétatif dans la mémoire rapide du calculateur pendant la résolution du problème. Les calculs ralentissent davantage pour la raison que, entre les différentes étapes de la résolution du problème, la commande passe à l'interprétateur.

On peut combiner la compilation avec l'interprétation. Le procédé consiste en ce que le programme généré contient des codes spéciaux appelés macro-instructions. Les macro-instructions s'interprètent lors de l'exécution par la machine du programme déjà généré. Une autre combinaison possible est basée sur l'interprétation : les parties interprétées du programme sont conservées dans la mémoire rapide, si elle le permet, et sont incluses dans l'information initiale sur le programme, à la place des groupes de codes qui correspondent à ces parties.

Chacune des méthodes (compilation et interprétation) a ses avantages, mais la méthode d'interprétation est plus souple. Par exemple, on peut prévoir dans le programme les ordres de formation de macro-instructions dans les cas où l'on ne connaît pas à l'avance le sous-programme qui sera nécessaire à la résolution du problème, etc.

De plus, la méthode d'interprétation simplifie le problème de la distribution de la mémoire, bien que demande une grande dépense supplémentaire de mémoire pour garder le programme interprétateur. Dans la méthode de compilation, on dispose d'une plus grande capacité de la mémoire rapide, puisque le compilateur n'y est pas présent lors de la résolution du problème.

5.2.2.2. Principe modulaire. Les systèmes d'aides à la programmation des calculateurs de la dernière génération sont basés sur le principe dit modulaire.

On appelle *modules* les « morceaux » des algorithmes donnés dans le langage du système exécutif ou dans un langage de programmation externe pour lesquels les conditions suivantes sont satisfaites:

1) Les « morceaux » des programmes donnés en langage algorithmique du système exécutif doivent être munis d'indications, permettant d'en construire un programme dans le langage du système exécutif, après avoir traité ces morceaux d'une manière convenable.

2) Les « morceaux » des algorithmes donnés dans des langages de programmation externes doivent être munis d'indications supplémentaires permettant d'en obtenir (par traduction et un traitement adéquat) les modules dans le langage du système exécutif, vérifiant la condition 1).

Le principe modulaire consiste à « monter » les programmes dans le langage du système exécutif à partir des modules. Les modules peuvent constituer une bibliothèque.

Pour monter un programme à partir des modules, il faut donner des précisions sur la manière de réunir les modules nécessaires. Le programme est généré automatiquement, à l'aide d'un programme spécial (générateur) prévu dans le software du calculateur. Remarquons que le programme obtenu n'est plus un module. Pour qu'il devienne module, il faut le munir de l'information nécessaire indiquée dans la condition 1) de la définition du module.

Le principe modulaire permet d'utiliser des modules composés dans les langages algorithmiques différents. Cela fournit des programmes objets d'une meilleure qualité, puisque chaque langage algorithmique est orienté vers un certain type d'applications. Le fait de conserver les modules et de les utiliser ensuite chaque fois qu on en a besoin permet d'épargner au programme le travail inutile.

Le principe modulaire représente un développement du principe de bibliothèque des sous-programmes

5.2.2.3. Bibliothèque de sous-programmes. La notion de bibliothèque de sous-programmes est très importante. Elle figure dans la définition même du software. Tous les éléments du système doivent être éléments d'une bibliothèque. De plus, dans le système d'exploitation il y a un programme bibliothécaire qui, sur l'ordre du dispatcher, appelle les sous-programmes d'exécution des macro-opérations. Pour cela le programme de travaux à exécuter doit comporter des macro-instructions. Ainsi, la bibliothèque des programmes qui englobe tout le software contient un programme bibliothécaire et une bibliothèque des macro-instructions qui lui correspond. En outre, le software d'un calculateur peut contenir d'autres bibliothèques et leurs programmes de service.

On appelle bibliothèque de programmes (de sous-programmes) une collection des programmes (de sous-programmes) composés à l'avance, pourvus chacun d'une information supplémentaire

permettant de les reconnaître. Les données sur tous les programmes (sous-programmes) doivent être réunies dans une table appelée catalogue. Celui-ci doit permettre de retrouver le programme: 1) d'après son nom, 2) d'après sa destination. Habituellement, on réunit dans une bibliothèque les programmes composés et présentés d'une manière spéciale. De plus, pour chaque bibliothèque, il faut posséder un programme d'extraction des programmes ainsi qu'un nombre de programmes de service (qui permettent d'y introduire de nouveaux programmes ou d'en exclure des programmes devenus inutiles). Il est commode de rapporter ces programmes de service à la bibliothèque, en les mettant par exemple à la première place dans le catalogue. Les programmes d'une bibliothèque peuvent être composés en codes symboliques que l'on peut convertir en langage machine, à condition que l'on donne les valeurs de certains paramètres. On dit de tels programmes qu'ils demandent la mise en œuvre d'un procédé d'adaptation, réalisé généralement au moyen des programmes spéciaux qu'on met également dans la bibliothèque.

Les programmes nécessitant l'adaptation, en plus de précisions générales, doivent être munis de l'information permettant justement cette adaptation. Les modules représentent des exemples de sous-programmes demandant l'ajustage et munis de l'information correspondante. Notons que les modules, étant des sous-programmes, n'appartiennent en général pas à la bibliothèque (ils peuvent ne pas figurer dans le catalogue). On peut utiliser les bibliothèques des sous-programmes aussi bien lors de l'interprétation (comme c'était le cas des macro-opérations) que lors de la compilation.

Remarquons qu'un système d'aides à la programmation peut contenir une bibliothèque des modules. Les éléments de cette bibliothèque, i.e. les modules composés en langage du système exécutif, sont utilisés par le programme générateur pour la génération du programme objet. Les modules sont une sorte de sous-programmes présentés d'une manière spéciale. Le compilateur « ajuste » les modules au cours de la compilation.

5.2.2.4. Programmation orientée vers un type de problèmes. La programmation basée sur l'utilisation d'aides à la programmation de haut niveau (i.e. de divers traducteurs, sauf, éventuellement, celui de l'autocode 1 : 1 et du compilateur de modules) s'appelle programmation orientée vers les problèmes, contrairement à la programmation des systèmes (§ 5.4). On entend par là la mise à la disposition du client de tous les moyens de programmation proposés par le software. Lorsqu'un problème ne demande pas de calculer une quantité énorme de variantes, on admet que son programme ne soit pas très « bon », c'est-à-dire qu'on admet l'éventualité d'une modification réduisant le temps de machine; en revanche on économise sur les frais et les efforts que nécessite la composition du programme,

de sorte que les frais totaux de programmation et de résolution du problème par le calculateur diminuent relativement à un programme « meilleur ». Le programmeur n'a généralement pas à s'occuper de l'organisation détaillée du programme, en se contentant de suivre la marche d'idées mises dans les traducteurs. Le programmeur de cette orientation peut même ignorer les particularités non essentielles du calculateur qu'il exploite, mais il approfondit ses connaissances des particularités des traducteurs dont il se sert, étudie un nombre de plus en plus grand des langages différents dont les traducteurs sont inclus dans le software de la machine. Au fur et à mesure du développement du software des calculateurs et du perfectionnement des langages et des traducteurs, le programmeur s'éloigne des questions qui, historiquement, étaient les plus difficiles dans la programmation (voir chapitre 2).

Il est vrai qu'on rencontre dans la pratique d'un centre de calcul, bien que rarement, des problèmes pour lesquels les méthodes de programmation orientée vers les problèmes ne marchent pas. C'est ou bien un problème à un nombre très grand de variantes, ou bien un problème qui exige une organisation du programme radicalement différente de celle qu'on a mise dans les traducteurs. On est alors obligé de composer le programme dans l'autocode 1 : 1. Pour la rédaction de tels programmes, le centre de calcul doit employer des programmeurs de très haute qualité, qui sont en réalité préparés à la programmation des systèmes.

5.2.3. Annexes. Maintenance du software en état de service. Classons parmi les annexes les programmes ou complexes de programmes destinés à résoudre les problèmes de type courant. Les annexes sont d'ordinaire groupées en *paquets d'annexes*. Chaque paquet contient les programmes se rapportant à la même classe de problèmes. Il est d'usage de former les paquets des problèmes de la statistique mathématique, de l'algèbre linéaire, de la programmation linéaire, etc. Les problèmes d'un paquet demandent souvent un « ajustage » avant le traitement. Il s'agit de donner certains paramètres, nécessaires au programme de service spécial (contenu dans le même paquet), pour modifier le programme de travail en vue de son utilisation future. Par exemple, dans le cas des problèmes de l'algèbre linéaire, ce paramètre est le nombre d'équations linéaires du système à résoudre.

Le nombre de paquets d'annexes caractérise la richesse du software du calculateur. Un paquet d'annexes (autrement, un *complexe de programmes d'application*) n'est pas une collection de programmes disparates, mais représente un ensemble de programmes interdépendants et munis souvent d'une structure assez compliquée. Il comprend en général un traducteur d'un langage orienté vers une classe étroite de problèmes, un programme de gestion, etc. Il n'est

pas rare qu'il contienne, pour la raison d'efficacité, des moyens modifiant le système d'exploitation.

Pour terminer la description du software du calculateur, arrêtons-nous encore sur les moyens permettant de maintenir le système à l'état de service. Il s'agit de programmes de duplication de l'information sur les supports mécanographiques, de programmes de constitution de bibliothèques, de programmes de « maintenance quotidienne » du système d'exploitation (nettoyage des bandes, tambours et disques magnétiques, édition des données, etc.). On rapporte au même type les programmes de génération du système d'exploitation. La génération peut être nécessaire : a) au début de la période d'exploitation du calculateur (formation d'un exemplaire du système d'exploitation correspondant à la configuration du calculateur et à son équipement) ; b) au cours de l'exploitation du calculateur (modifications du système d'exploitation liées à celle de l'équipement du calculateur ou de son mode d'exploitation, ou à la suite d'une modernisation du système).

Remarquons que le software du calculateur réunit un grand nombre de programmes et se compose de dizaines de milliers d'ordres. Il n'y a pas à s'étonner qu'un tel système de programmes ait des défauts ou qu'on tombe de temps en temps sur une erreur qu'on n'a pas décelée avant. Il est naturel que les systèmes s'améliorent au fur et à mesure de l'exploitation. En outre, de grandes modernisations, dues à la nécessité d'y incorporer de nouveaux éléments, par exemple, de nouveaux traducteurs, sont à envisager.

On voit que le programme générateur représente un élément essentiel du software, et que son utilisation n'est pas si rare.

§ 5.3. Le software du calculateur comme interprétation de certaines notions de la théorie des algorithmes

On a expliqué au p. 1.5.6 qu'un calculateur représente, du point de vue de la théorie des algorithmes, une réalisation physique de l'algorithme d'exécution d'un programme. Les programmes pour ce calculateur représentent une famille d'algorithmes donnés en langage algorithmique de la machine. Les données initiales et les résultats sont des constructions du langage des opérandes correspondant à la machine donnée.

En désignant un programme par t, une donnée initiale par s, l'algorithme d'exécution par W et le résultat par r, nous avons écrit

$$\widetilde{W}(t;\ s) = r, \tag{5.1}$$

où $\widetilde{W}$ désigne l'application engendrée par l'algorithme W, et $t;\ s$ représente une construction formée par le programme et la donnée initiale.

Il est possible de construire, dans un langage algorithmique, un algorithme D, applicable à des couples τ, σ et vérifiant la relation

$$\tilde{D}(\tau, \sigma) = r \tag{5.2}$$

On définit ainsi une nouvelle classe d'algorithmes $\{\tau\}$. En substituant dans (5.1) la construction D à t et les constructions τ, σ à s on obtient

$$\tilde{W}(D; \tau, \sigma) = r \tag{5.3}$$

En d'autres termes, comme D est constant, on peut dire que l'algorithme W, pour D donné, représente un nouveau algorithme W' tel que

$$\tilde{W}'(\tau, \sigma) = r. \tag{5.4}$$

On peut dire qu'ayant introduit, à la place de l'algorithme W et de la machine qui lui répond, l'algorithme D et deux nouveaux langages: le langage algorithmique $\{\tau\}$ et le langage des données initiales $\{\sigma\}$, on a obtenu une nouvelle machine W' qui résout le même problème mais au moyen d'un autre programme et dans une autre représentation des données initiales.

Ainsi, en établissant un programme (l'unique et toujours le même au moment où l'on commence à utiliser la machine W), nous transformons cette machine en une nouvelle machine W', avec un nouveau langage algorithmique et un nouveau langage des opérandes.

Cette idée est à la base de systèmes d'exploitation. Dans notre exemple, les modifications qu'a subies le système d'exploitation ne concernent que les langages (d'algorithmes et des opérandes). Si les nouveaux langages sont plus commodes que ceux du calculateur W, alors l'emploi d'un tel système d'exploitation est déjà avantageux. Or, on peut aboutir à de meilleurs résultats si l'on arrive à élaborer un tel programme D que:

a) l'opérande s prenne la forme

$$s = \tau_1, \sigma_1; \tau_2, \sigma_2; \ldots; \tau_n, \sigma_n, \tag{5.5}$$

où $\tau_1, \tau_2, \ldots, \tau_n$ sont des phrases du langage $\{\tau\}$, et $\sigma_1, \sigma_2, \ldots$ $\ldots, \sigma_n$, celles du langage $\{\sigma\}$;

b) le résultat r satisfasse en même temps à la condition

$$r = r_1, r_2, \ldots, r_n, \tag{5.6}$$

où $r, r_1, r_2, \ldots, r_n$ sont des phrases du langage $\{r\}$;

c) pour un algorithme V on ait

$$\tilde{V}(\tau_i, \sigma_i) = r_i. \tag{5.7}$$

Alors, compte tenu de (5.1), (5.5) et (5.6), on aurait

$$\tilde{W}(D; \tau_1, \sigma_1; \ldots; \tau_n, \sigma_n) = r_1, r_2, \ldots, r_n, \tag{5.8}$$

ce qui signifie qu'une application du calculateur W, équipé du programme D fixe, à l'opérande de la forme (5.5) est équivalente à l'application du calculateur V, pour $i = 1, 2, \ldots, n$, au programme τ_i, avec la donnée initiale σ_i.

En d'autres termes, on obtiendrait un programme D qui permettrait de résoudre n problèmes dont les programmes sont exprimés dans le langage $\{\tau\}$.

Le programme D n'est rien d'autre qu'un système d'exploitation mis dans le calculateur W et le faisant produire autant de résultats qu'il est possible d'obtenir avec n calculateurs V, ou avec un calculateur V en n passes.

Il ne vaut pas la peine de démontrer cet effet pour de nombreux calculateurs existants, l'existence même des dispatchers pour ces calculateurs en représente la meilleure preuve. On n'est pas certain, il est vrai, qu'il soit possible de construire un dispatcher pour tout calculateur pensable. Or, ce n'est pas que l'on cherche à ce moment.

Le dispositif réalisant les interruptions permet de nombreuses simultanéités de fonctionnement. Dans un calculateur disposant de plusieurs processeurs, cette possibilité est encore plus large. En nous bornant au cas de calculateur à un processeur, nous nous tirons d'affaire avec la seule formule (5.8), car une telle machine exécute simultanément seules les actions dont l'ordre d'exécution n'a pas d'importance, de sorte qu'un travail parallèle peut être remplacé par un travail dont les étapes s'exécutent dans un quelconque des ordres admissibles.

Ainsi, un système d'exploitation représente un algorithme d'exécution D composé dans le langage algorithmique du calculateur et satisfaisant à la condition (5.8).

Le cas de calculateur multiprocesseur demande une généralisation de la notion d'algorithme que nous ne considérons pas ici.

L'essentiel de certains autres éléments du software, tels que les traducteurs (qui se rapportent au système d'aides à la programmation) et les programmes d'entrée de l'information, n'est pas plus difficile à comprendre.

On peut imaginer un algorithme T et un langage $\{a\}$ tels que

$$\widetilde{T}\,(a) = \tau, \tag{5.9}$$

où τ est un programme dans le langage du système exécutif. Si encore T est lui-même un algorithme dans le langage du système exécutif, i.e. appartient au langage algorithmique $\{\tau\}$, alors le même système exécutif composé de W et D permet d'obtenir le programme τ au moyen du couple T, a. L'algorithme T vérifiant (5.9) s'appelle traducteur, et le langage $\{a\}$, langage algorithmique d'entrée.

Il est logique, de construire, par analogie avec l'algorithme T, un algorithme B dans le langage $\{\tau\}$, qui donnerait

$$\widetilde{B}(b) = \sigma. \qquad (5.10)$$

Un tel algorithme transformerait les données initiales présentées dans un langage des opérandes $\{b\}$ en les données initiales dans un langage des opérandes $\{\sigma\}$.

Si les langages $\{a\}$ et $\{b\}$ sont plus commodes pour les programmeurs que ceux du système exécutif, il devient possible de rédiger le programme dans le premier et d'exprimer les données initiales dans le second, puis les transformer, au moyen du système exécutif, respectivement en l'algorithme et les données initiales pour ce système.

On aurait pu, par analogie, construire un algorithme C dans le langage $\{\tau\}$, qui traduirait les mots du langage $\{r\}$ en un langage $\{c\}$, i.e. fournirait

$$\widetilde{C}(r) = c. \qquad (5.11)$$

Un tel algorithme permettrait de traduire les résultats de résolution des problèmes par le système exécutif en un nouveau langage. Ce serait utile dans le cas où le langage des résultats $\{c\}$ se trouverait plus commode pour les utilisateurs que le langage $\{r\}$.

L'algorithme C représente le programme de sortie des résultats du calculateur. Nos considérations nous amènent, d'une façon inattendue, à une analogie entre le traducteur, le programme d'entrée et le programme de sortie. Donc, en faisant la classification des programmes du software, il serait plus logique d'y distinguer les parties autres que celles que nous avons évoquées au § 5.2. Il faudrait en dégager les systèmes suivants:

1. Système d'exploitation.
2. Système d'aides à la programmation.
3. Système d'entrée.
4. Système de sortie.
5. Annexes.
6. Système de maintenance du software à l'état de service.

Nous avons classé parmi les aides à la programmation le programme générateur (à partir des modules). Ce programme représente un cas spécial de traducteur, et son rôle s'explique complètement par la formule (5.9).

Pour conclure ce paragraphe, faisons encore une remarque. En ce qui concerne la programmation, la pratique devançait toujours la théorie. L'idée du software ne fut pas le résultat d'une analyse théorique des notions de la programmation, mais fut née de trouvailles heureuses des programmeurs, qui faisaient plutôt la preuve de leur talent d'ingénieurs que d'une formation mathématique solide. Certains auteurs sont assez audacieux pour affirmer que le développe-

ment ultérieur des systèmes sera lié de plus en plus intimement à la recherche théorique. Evidemment, cela exige que les éléments de la théorie des algorithmes dont se servira la théorie de la programmation soient revus et adaptés de façon à permettre une position correcte et une étude exhaustive de ses problèmes. Or, actuellement, la théorie des algorithmes ne présente pas encore une telle possibilité.

§ 5.4. Programmation des systèmes

On appelle *programmation des systèmes* le travail de constitution des programmes du software contrairement à la programmation orientée vers certains types de problèmes (voir p. 5.2.2). Il est évident qu'on doit rapporter à la programmation des systèmes d'établissement de systèmes de programmes pour certains problèmes qu'on n'arrive pas à résoudre avec la programmation des problèmes.

La programmation d'un système doit commencer par l'établissement de la structure de la bibliothèque de programmes B et de deux ensembles de règles : F, qui détermine la structure des programmes et Φ, qui détermine la documentation d'accompagnement des programmes et des groupes de programmes. Dans ce travail il faut tenir compte de la composition du software du calculateur (§ 5.3).

A l'étape suivante, on dresse une liste détaillée des programmes avec indication de leurs fonctions. La liste des programmes est établie compte tenu du mode d'exploitation du calculateur ou de l'ensemble de calculateurs. On choisit les langages de programmation, ceux des données initiales et des résultats.

Toute la première étape (qui précède la rédaction des programmes concrets) peut être appelée *étape de l'avant-projet*.

L'avant-projet est à la base du travail de constitution du software du calculateur. Au cours de la rédaction des programmes concrets, on y apporte des modifications et des corrections, de sorte que lorsqu'on termine les programmes du premier cycle, on en obtient également une description qui peut être appelée *projet d'exécution*. Nous voyons que l'établissement du projet et l'élaboration du système de programmes se font parallèlement. Cela est dû au fait que la théorie des systèmes est si peu développée qu'il n'a aucun espoir de faire précéder l'élaboration des programmes de calculs théoriques raisonnables.

Les programmes du système doivent être d'une qualité particulièrement haute, c'est-à-dire qu'ils ne doivent pas requérir pour leur implantation de trop grandes capacités de mémoire et doivent s'exécuter en un temps acceptable. Si le fonctionnement des programmes demande un grand volume des mémoires du calculateur, cela réduit la place que peuvent occuper les programmes d'utilisateurs. D'autre part, si les programmes du système dépensent trop de temps, cela augmente les pertes constantes du temps de la machine durant

l'exploitation du système. Ainsi, en établissant les programmes du système, il faut économiser la mémoire d'une part et le temps de la machine d'une autre. On sait bien que ces deux conditions sont contradictoires. Pour cette raison, en élaborant les programmes d'un système on emploie une technologie spéciale dont les traits essentiels nous décrivons dans ce qui suit.

Lorsqu'un calculateur n'a pas encore de système, on élabore l'autocode le plus simple possible qui représente, par rapport au langage algorithmique du calculateur, un autocode 1 : 1. Pour cet autocode on programme un traducteur qui ne nécessite aucun système d'exploitation pour être utilisé. Ce genre de traducteur est habituellement appelé assembleur de niveau inférieur, l'autocode lui-même étant appelé langage d'assemblage de niveau inférieur.

La programmation de l'assembleur se fait ou bien directement dans le langage de la machine donnée, ou bien à l'aide d'une autre machine qui possède déjà un système.

On n'exige de cet assembleur ni une rapidité de fonctionnement, ni une économie de mémoire, car il est employé uniquement pour la création des programmes du système dont il ne fera pas partie.

On demande seulement de cet assembleur qu'il permette de générer n'importe quel programme dans le langage du calculateur à partir d'un langage d'entrée assez commode pour le programmeur pour que le nombre d'erreurs que celui-ci peut commettre soit assez réduit et éliminable au cours de la mise au point du programme. On se sert de l'assembleur de niveau inférieur pour l'établissement de programmes de service du système.

La programmation ultérieure se fait en employant l'assembleur donné. On crée avant tout le système d'exploitation (le dispatcher et le superviseur). Puis on réalise une nouvelle version d'assembleur qui permet de traduire les algorithmes initiaux non plus dans le langage de la machine, mais dans celui du système exécutif. Les autres programmes du système sont ensuite établis à l'aide de ce nouvel assembleur (dont le langage d'entrée est différent de celui du premier assembleur).

Remarquons que le second assembleur se distingue du premier non seulement par le fait qu'il est basé sur le système d'exploitation déjà créé, mais encore par ce qu'il n'utilise pas les instructions privilégiées, qui ne sont admissibles que pour le dispatcher et donc les programmes qu'il permet de générer ne les contiennent pas. Autrement dit, par rapport au système d'exploitation, le second assembleur représente un programme ordinaire, tandis que le premier n'étant limité en rien, a pu servir à la création du système d'exploitation.

La programmation dans le langage du calculateur (si l'on recourt encore à cette méthode), puis dans le langage du premier assembleur et enfin dans le langage du deuxième assembleur (qui appartient

déjà au système d'aides à la programmation), se réalise par la méthode opératorielle dont les éléments ont été décrits au § 3.4.

En d'autres termes, pour tout problème, on compose d'abord son algorithme de résolution en YALS, puis on l'adapte, au moyen des transformations équivalentes, aux particularités du calculateur ou (plus tard) du système exécutif, ensuite, en général, mentalement et parallèlement au codage des opérateurs dans le code de la machine ou dans le langage du premier ou du deuxième assembleur, on met l'algorithme sous une forme qui correspond aux particularités des instructions de la machine ou des opérateurs du langage d'assembleur. Le programme qu'on obtient se trouve présenté soit dans le langage de la machine s'il s'agit de créer le premier assembleur, soit dans le langage d'assemblage dans le cas de tous les autres programmes. Les programmes obtenus dans le langage d'assemblage sont ensuite traduits dans le langage du calculateur.

Les programmes ainsi obtenus sont minutieusement corrigés et essayés en machine. Notons que lors de la rédaction des programmes, on tient compte des exigences imposées par l'ensemble de règles F.

Lorsque le programme est mis au point, on établit pour lui toute la documentation demandée par les règles Φ, et on le range dans la bibliothèque principale du système ou dans une section correspondante.

Les problèmes d'utilisateur que l'on n'arrive pas à programmer moyennant les éléments de niveau supérieur du système d'aides à la programmation, sont programmés dans le langage du deuxième assembleur par la méthode opératorielle.

§ 5.5. Notions supplémentaires sur les langages formels

5.5.1. Grammaires formelles. Nous avons dit au § 1.2 qu'un langage formel est donné dès qu'est donné son métalangage syntaxique. Nous avons séparé la notion de sémantique de celle de langage formel, en admettant (au p. 1.2.2) des langages formels à sémantique vide. En d'autres termes, nous avons adopté la

D é f i n i t i o n I. On appelle *langage formel* l'ensemble de constructions déterminé par un métalangage syntaxique.

On suppose ici que le métalangage syntaxique détermine d'une manière univoque l'alphabet des lettres et celui des liaisons du langage objet et contient les règles de construction des phrases de ce langage. Il faut que le nombre des règles soit fini, et que chaque règle admette l'application des opérations appartenant à un ensemble fini d'opérations (une règle admet des combinaisons et des répétitions des opérations correspondantes).

D é f i n i t i o n II. On appelle *langage formel muni d'une sémantique* un langage formel que l'on considère avec un ensemble

d'éléments et une application qui, à toute phrase de ce langage fait correspondre d'une manière biunivoque un élément de l'ensemble mentionné.

L'ensemble dont il s'agit dans la définition II sera dit *sémantique*. Il se peut que, parmi ses éléments, il y ait un objet vide. Notons qu'une modification de l'ensemble sémantique ou de l'application mentionnée est équivalente à la construction d'un nouveau langage formel muni d'une sémantique.

Certaines formes de métalangages syntaxiques sont appelées *grammaires formelles*.

Nous allons étudier deux types de grammaires formelles génératives qu'on appelle respectivement *grammaires inductives* et *grammaires déductives*. Les premières sont basées sur la notion de métalangage inductif dont nous avons abordé la description au p. 1.2.2; la première description complète des grammaires inductives fut donnée dans [28]; les secondes qui historiquement précèdent les premières, furent introduites dans les travaux de Noam Chomsky.

5.5.2. Langages formels engendrés par des grammaires inductives. Il est commode d'attribuer à chaque métasymbole désignant une construction concrète (un morphème) le nom d'une opération de rang nul dont les données initiales sont des collections vides de constructions et dont les valeurs sont les constructions désignées par ce métasymbole (voir p. 1.5.3 et l'exemple 1.28 du p. 1.5.4).

On définit une *grammaire générative inductive* par les données suivantes: trois alphabets, une collection finie de métaformules, un ensemble fini d'opérations et une lettre spéciale σ:

$$\Gamma = (A, B, C, F, \Omega, \sigma),$$

où

1. A et C sont des alphabets disjoints de lettres.
2. B est un alphabet des liaisons (des connecteurs).
3. σ est une lettre dans C.
4. F est un ensemble fini de métaformules qui sont des mots dans C et sont, pour fixer les idées, de la forme:

$$\alpha_{t_i} ::= f_i(\alpha^i_{s_1}, \alpha^i_{s_2}, \ldots, \alpha^i_{s_r});$$

où les α_{t_i}, $\alpha^i_{s_1}$, . . ., $\alpha^i_{s_r}$. f_i, le symbole $::=$ de même que les parenthèses, le point-virgule et la virgule sont des mots dans C. Le symbole f_i est le nom d'une opération de Ω. Si f_i et f_j sont les noms d'une même opération, ils représentent la même lettre dans C.

5. Ω est un ensemble fini d'opérations ω_i dont les données initiales sont des collections de constructions de la classe (A, B) et dont les résultats sont des constructions de la même classe. On suppose que Ω contienne un nombre (non nul) d'opérations de rang

nul dont chacune est identiquement égale à une construction de la classe (A, B).

6. A toute opération ω_i de Ω il correspond au moins une formule dans F qui contient le nom de cette opération dans sa partie droite.

L'ensemble des valeurs des opérations de rang nul appartenant à F s'appelle *base* de la grammaire (et aussi du langage objet).

Les lettres de l'alphabet C qui sont utilisées dans les formules comme arguments et dans leurs parties gauches s'appellent *symboles non terminaux*. En particulier, σ est un symbole non terminal. Les symboles non terminaux désignent des classes de constructions produites lors de la génération du langage objet (ce qu'on verra plus bas).

Introduisons la notion d'*inductibilité*. Etant donné les ensembles d'opérations Ω et de métaformules F nous dirons que β est *directement inductible* à partir des éléments d'un ensemble M, si β est ou bien un élément de M, ou bien s'obtient à partir des éléments de M au moyen d'une métaformule appartenant à F.

Ensuite, β est *inductible* (tout court) à partir des éléments d'un ensemble M, si ou bien il est directement inductible de ces éléments, ou bien s'obtient par applications successives des formules de F aux éléments de l'ensemble M et aux résultats de ces applications.

Nous allons appeler langage formel engendré par une grammaire inductive l'ensemble des constructions σ de la classe (A, B) (i.e. portant le nom σ) qui sont inductibles des éléments de la base. C'est-à-dire qu'une construction K de la classe (A, B) appartient à un langage formel L que l'on veut définir lorsqu'elle est inductible à partir de certaines valeurs des opérations de rang nul et s'obtient par une formule dont le membre gauche représente la lettre σ. On peut le noter comme suit:

$$L = \{\sigma \mid \sigma \leftarrow H\},$$

où H est la base.

Le langage L engendré par une grammaire inductive est le langage objet; A et B sont respectivement les alphabets des lettres et des liaisons du langage objet; H est l'ensemble de ses morphèmes (voir p. 1.2.1). Nous avons fait correspondre à ces morphèmes les noms des opérations de rang nul dans la grammaire générative inductive, grâce à quoi toutes les métaformules ont pris la forme des mots dans C. L'alphabet C est une liste de métasymboles; F est le métalangage que nous avons déjà vu au p. 1.2.1 (avec des modifications non essentielles: les morphèmes du langage objet dans les parties droites des métaformules sont remplacés par les métasymboles spéciaux; la lettre σ désigne le métasymbole distingué correspondant à la métanotion de « phrase ».

Un cas spécial de langage formel engendré par une grammaire inductive est représenté par le langage formel des « chaînes de let-

tres », où l'alphabet B est celui des liaisons d'enchaînement. Dans ce cas il est superflu de mentionner B, et la grammaire prend la forme

$$\Gamma = (A, C, F, \Omega, \sigma),$$

où A, C sont des alphabets disjoints, σ une lettre dans C, les opérations ω_i appartenant à Ω sont des opérations sur les collections ordonnées de *mots* dans A, de plus, les opérations de Ω et les formules de F sont liées de la manière décrite plus haut.

Nous avons mentionné au § 1.2 que la notation normale de Backus représente, d'un certain point de vue, un cas spécial de métalangage syntaxique générateur par récurrence. Il en résulte que chaque langage formel donné au moyen de la notation normale de Backus peut être engendré par une grammaire inductive.

La génération par récurrence d'un langage formel ressemble beaucoup à la construction des phrases dans la plupart des langues naturelles, notamment dans le français. On voit là l'intérêt des grammaires inductives.

5.5.3. Langages formels engendrés par des grammaires déductives. Soient A et C deux alphabets disjoints de lettres; σ, un symbole de l'alphabet C; B et D, deux alphabets de liaisons tels que $B \subseteq D$. Soit, de plus, Ω un ensemble fini des opérations $\omega_1, \omega_2, \ldots, \omega_n$ dont les données initiales et les résultats sont (voir § 1.1) des constructions de la classe $(A \cup C, D)$.

L'ensemble

$$G = (A, B, C, D, \Omega, \sigma)$$

s'appelle *grammaire générative déductive*.

Définissons la notion de déductibilité.

Si x et y sont des constructions de la classe $(A \cup C, D)$ et si

$$y = \omega_i(x),$$

nous dirons que y est *immédiatement déductible de* x et nous noterons

$$x : \Rightarrow y.$$

Nous dirons que y *est déductible de* x et noterons $x \Rightarrow y$ si l'une des relations suivantes a lieu:

1) $x : \Rightarrow y$;

2) $x \Rightarrow z$; $z : \Rightarrow y$.

En d'autres termes, $x : \Rightarrow y$ signifie que y s'obtient de x au moyen d'une seule opération ω_i, et $x \Rightarrow y$ signifie que y s'obtient de x au moyen d'une suite de telles opérations.

On appelle *langage formel* (ou *langage* tout court) engendré par une grammaire déductive G l'ensemble L des constructions de la

classe (A, B) déductibles de σ, c'est-à-dire $L = \{x \mid \sigma \Rightarrow x ; x$ appartient à la classe $(A, B)\}$.

Dans notre exposé L est le langage objet, A et B sont respectivement les alphabets des lettres et des liaisons que l'on peut utiliser dans les constructions du langage objet, et x $(x \in L)$ est une phrase du langage objet L. L'alphabet C est une liste de métasymboles (servant à désigner les catégories grammaticales). La lettre σ est une lettre de l'alphabet C employée pour exprimer la métanotion de « phrase ». L'ensemble des règles Ω dont chacune indique une opération est un métalangage syntaxique. Un métalangage de ce type sera appelé *générateur par déduction*. L'alphabet D contient seules les liaisons qui sont employées dans le métalangage Ω ou appartiennent à l'alphabet B.

Dans le cas spécial où $B = D$ est un alphabet des liaisons d'enchaînement, on peut ne pas mentionner B et D.

On appelle *grammaire générative déductive* un ensemble

$$G = (A, C, \Omega, \sigma),$$

où A et C sont deux alphabets disjoints (de lettres), σ appartient à C. On appelle langage L engendré par la grammaire G l'ensemble des mots dans A ou, comme on dit parfois, des chaînes de lettres dans A, déductibles du mot monogramme σ. Dans ce cas L est un langage des mots.

Les définitions que nous venons de donner du langage L diffèrent des descriptions habituelles de langages par le fait que chaque mot (phrase) y est déduit du mot monogramme σ. Tandis qu'habituellement, pour former des mots et des phrases d'un langage, on part d'une collection de mots élémentaires qui servent à construire des mots de plus en plus compliqués à l'aide de règles de grammaire. A partir d'un certain niveau, on obtient les constructions qui sont des mots et des phrases du langage. C'est-à-dire, les grammaires de notre expérience sont non pas déductives, mais inductives, menant à partir des particules élémentaires aux constructions du langage.

Signalons un cas particulier important de grammaire générative déductive conduisant à un langage des mots. On appelle *substitution markovienne* dans l'alphabet $A \cup C$ l'opération, désignée par $P : \rightarrow Q$ où P et Q sont des mots dans $A \cup C$, qui consiste à trouver dans le mot proposé la première occurrence (à compter du début) du mot P et à la remplacer par le mot Q; ce qu'on obtient est le résultat de l'opération. Dans le cas où P ne figure pas dans le mot proposé, nous convenons que l'opération est inapplicable à ce mot. Convenons que le mot P n'est pas vide. Désignons sa première lettre par p.

Une collection de substitutions markoviennes de la forme

$$p : \rightarrow \alpha ; \quad P : \rightarrow Q ; \quad \alpha : \rightarrow p ;$$

s'appelle ***substitution*** et se désigne $P \rightarrow Q$ dans le cas où l'on sait que: a) α est une lettre dans l'alphabet C; b) cette lettre n'entre dans aucune autre substitution et ne figure pas dans les mots P et Q.

Au moyen d'une substitution (en utilisant de façon convenable les substitutions markoviennes dont elle se compose) on peut remplacer par Q non seulement la première, mais aussi la deuxième, la troisième, etc. occurrence du mot P dans le mot proposé. Une substitution n'est pas une opération, parce que pour les mots initiaux contenant plusieurs occurrences de P le résultat n'est pas univoque. Tout de même nous nous permettrons par la suite un abus de langage et appellerons les substitutions opérations. De plus, nous appellerons substitutions les désignations des substitutions.

On considère en général les grammaires génératives déductives où Ω représente un ensemble fini de substitutions.

Les grammaires de ce type et les langages qu'elles engendrent seront dit *normaux*.

La classe des langages normaux est très vaste *). Dégageons-en deux cas spéciaux les plus intéressants.

Un langage normal et une grammaire normale qui l'engendre sont appelés ***langage*** et ***grammaire des composantes directes*** (CD-***langage*** et CD-***grammaire***) si chaque substitution de la grammaire normale a la forme

$$u\xi v \rightarrow uyv,$$

où ξ est une lettre de l'alphabet C et u, v sont des mots dans C, tandis que y est un mot non vide dans $A \cup C$.

Un cas particulier des langages et grammaires normaux représentent les langages et les grammaires indépendants du ***contexte*** (« context free » dans la littérature de langue anglaise) (CF-***langages*** et CF-***grammaires***), où toutes les substitutions sont de la forme

$$\xi \rightarrow y,$$

ξ étant une lettre dans C et y, un mot dans $A \cup C$, éventuellement vide. Une CF-grammaire dont aucune substitution n'a le membre droit vide est un cas spécial de CD-grammaires.

5.5.4. Liens entre les grammaires inductives et déductives. *Pour tout langage L engendré par une grammaire inductive, il existe une grammaire générative déductive.*

Soit un langage L engendré par une grammaire inductive $\Gamma =$ $= (A, B, C, \Omega, F, \sigma)$; on peut construire la grammaire déductive G correspondante de la manière suivante.

Formons l'alphabet $D^* = B \cup \{\mapsto, \rightarrow, \rightarrow\!|\}$, où les symboles $\mapsto$, $\rightarrow$, $\rightarrow\!|$ désignent les liaisons de succession et ne figurent pas

---

*) La famille des langages normaux est isomorphe à la famille bien étudiée dans la logique mathématique des ensembles récursivement énumérables.

dans les alphabets A et C. Construisons ensuite l'ensemble des opérations Ω^* comme suit. Si F contient une formule

$$y = f(x_1, x_2, \ldots, x_n),$$

nous introduisons dans Ω la substitution

$$y \to f(x_1, x_2, \ldots, x_n)$$

ainsi que la substitution généralisée

$$„f(x_1, x_2, \ldots, x_n)" \to Rf(x_1, x_2, \ldots, x_n).$$

La dernière notation suppose que le symbole R n'entre pas dans les alphabets utilisés jusqu'à ce moment et signifie qu'il faut remplacer la notation de la fonction f, dans laquelle on a remplacé les arguments $x_1, x_2, \ldots, x_n$ par certaines constructions $\alpha_1, \alpha_2, \ldots, \alpha_n$ de la classe $(A \cup C, D)$, par le résultat d'exécution de l'opération $\omega(\alpha_1, \alpha_2, \ldots, \alpha_n)$ dont f est le nom.

Nous aboutissons ainsi à la grammaire déductive cherchée

$$G = (A, B, C, D^*, \Omega^*, \sigma).$$

Pour tout langage L engendré par une grammaire déductive il existe une grammaire générative inductive.

Soit L un langage engendré par la grammaire déductive

$$G = (A, B, C, D, \Omega, \sigma).$$

Ici $\Omega\ \{\omega_1, \omega_2, \ldots, \omega_n\}$, où les ω_i sont les opérations de rang un. Construisons une grammaire inductive engendrant le même langage L de la manière suivante. Choisissons un alphabet C^* n'ayant pas d'élément commun avec les autres alphabets. Supposons qu'il contienne les lettres $f_0, f_1, \ldots, f_{n+1}, \sigma^*, ::=, (,)$. Construisons : l'opération de rang nul $\bar{\omega}_0 \equiv \sigma$, les opérations de rang un que nous définissons, pour $i = 1, 2, \ldots$, par les conditions

$$\bar{\omega}_i(z) = \begin{cases} \omega_i(z), \text{ si } z \text{ n'est pas une construction} \\ \text{de la classe } (A, C), \\ \text{indétermination, si } z \text{ est une construction} \\ \text{de la classe } (A, C), \end{cases}$$

et enfin l'opération

$$\bar{\omega}_{n+1}(z) = \begin{cases} z, \text{ si } z \text{ est une construction} \\ \text{de la classe } (A, C), \\ \text{indétermination, si } z \text{ n'est pas une cons-} \\ \text{truction de la classe } (A, C). \end{cases}$$

Les nouvelles opérations que nous venons d'introduire sont bien conformes à la définition de l'opération formulée au § 1.4.

En nous servant les lettres de l'alphabet C^*, choisissons $f_0, f_1, \ldots, f_n, f_{n+1}$ pour noms respectifs des nouvelles opérations et

composons les formules

$$z = f_0,$$
$$z = f_i(z) \qquad (i = 1, 2, \ldots, n),$$
$$\sigma^* = f_{n+1}(z).$$

Désignons cet ensemble de formules par F^*. Posons $A^* = A \cup C$, $B^* = B \cup D$. Nous avons obtenu la grammaire inductive cherchée

$$\Gamma = (A^*, B^*, C^*, \Omega^*, F^*, \sigma^*)$$

pour le même langage L.

L'assertion suivante découle des propositions démontrées.

La classe des langages engendrés par des grammaires inductives et celle des langages engendrés par des grammaires déductives se confondent.

Remarquons que, s'il y a deux langages $L' = \{s\}$ et $L'' = \{t\}$, alors l'ensemble des éléments (s, t) peut être considéré comme un nouveau langage, car la formation du couple s, t représente la production d'une construction en joignant deux constructions particulières s et t par une nouvelle liaison de succession. Pour ce nouveau langage il est aisé de construire une grammaire générative.

Signalons un cas spécial important de la dépendance indiquée plus haut entre les grammaires génératives inductive et déductive.

Si, dans une grammaire linéaire *) inductive $\Gamma = (A, B, \Omega, F, \sigma)$, l'ensemble d'opérations Ω ne contient, outre les opérations de rang nul, que les *opérations de concaténation*, alors pour le langage L engendré par la grammaire inductive Γ il existe une FC-grammaire déductive. Réciproquement, si un langage L est indépendant du contexte, alors il existe une grammaire inductive Γ dont l'ensemble d'opérations Ω ne contient, outre les opérations de rang nul, que des opérations de concaténation.

L'équivalence des classes des langages engendrés par des grammaires inductives et déductives nous permet de choisir à notre gré le type de grammaire.

5.5.5. Sémantique formelle d'un langage formel. Soit $L = \{x\}$ un langage. Nous aurons besoin de la notion de *sous-langage*. Nous appelons sous-langage un sous-ensemble de langage. Puisqu'en général, un langage n'est pas un ensemble fini de mots, il faut être prudent en définissant un sous-langage. Supposons que $L' = \{z\}$ soit un autre langage formel. S'il existe (voir p. 1.5.5) un algorithme $W(x, z)$ applicable à certains couples x, z, alors l'ensemble des x qui figurent dans au moins un de ces couples s'appelle sous-lanagage du langage L. En particulier, le domaine d'applicabilité d'un algorithme quelconque est un sous-langage.

---

*) i.e. dans la grammaire d'un langage des mots.

Nous définissons la notion de sémantique formelle d'un langage L d'une telle manière qu'elle corresponde à la notion intuitive de sémantique.

Soit donc Λ un sous-langage du langage L. Nous disons qu'une sémantique formelle est donnée sur Λ si l'un des cinq cas suivants se présente :

1. Une sémantique formelle est donnée sur un sous-langage Λ^* d'un langage L^*, et l'on connaît un algorithme T tel que $T(x) = y$, où $x \in \Lambda$, $y \in \Lambda^*$.

On dit alors que la sémantique est donnée par la méthode de traduction.

2. Il existe un ensemble

$$B = \{b\}$$

et un algorithme U tel que $U(x) = b$, où $x \in \Lambda$, $b \in B$.

Dans ce cas le mot x s'appelle *nom* de l'élément b, et celui-ci *dénotation* du nom x.

3. Il existe un langage $L' = \{z\}$, et un algorithme $W(\alpha, \beta)$ tel que $W(x_i, z_{i,j}) = y_{i,j}$, avec $x_i \in \Lambda$, $z_{i,j} \in L'$, $y_{i,j} \in \Lambda''$, où Λ'' est un sous-langage du langage L.

On dit alors que la sémantique sur Λ est donnée intérieurement par rapport à L.

4. Il existe un langage $L'' = \{z\}$ et un algorithme $W(\alpha, \beta)$ tel que $W(x_i, z_{i,j}) = r_{i,j}$, où $x_i \in \Lambda$, $z_{i,j} \in L''$, $y_{i,j} \in L''$.

Dans ce cas, à chaque x_i il correspond (du point de vue de la sémantique) un objet complexe qui représente un ensemble de couples de la forme $(z_{i,j}, y_{i,j})$, et qui peut être interprété comme une application de l'ensemble $Z_i = \{z_{i,j}\}_{j=1}^{\rho}$ sur l'ensemble $Y_i = \{y_{i,j}\}_{j=1}^{\rho}$.

5. Le sous-langage Λ représente une somme d'un nombre fini de sous-langages qui possèdent chacun sa sémantique donnée par l'une des façons précédentes.

Nous rencontrons dans la pratique toutes les méthodes énumérées de définir une sémantique. Par exemple, nous nous servons de la première méthode dans la création des langages artificiels. La sémantique des mots d'un langage artificiel peut être donnée à l'aide d'un dictionnaire et d'un système de règles de traduction permettant de passer des phrases du langage artificiel à celles d'une langue naturelle. Telle peut être la sémantique de l'espéranto.

La deuxième méthode s'applique à un sous-langage qui représente un ensemble des noms propres (par exemple, un sous-langage de ce type de la langue française contient les mots « Paris », « Lune », « Mont-Blanc », « Baudelaire »).

Quant à la troisième méthode de donner une sémantique, il s'agit d'établir des liaisons entre certains mots et certaines classes de mots. Choisissons, par exemple, dans la langue française le mot « maison »

et lui attachons une famille de mots qui expriment cette notion : « isba », « cabane », « immeuble », « gratte-ciel », « villa », etc., ou bien établissons des rapports entre le mot « maison » et l'ensemble des mots « sous-sol », « étage », « grenier », « cage d'escalier », « rez-de-chaussée », etc., ou enfin entre le même mot « maison » et le mot (phrase) « construction se composant d'un fondement, d'un toit, d'un sous-sol, d'un rez-de-chaussée, d'étages et d'une cage d'escalier » (dans ce dernier cas l'ensemble de mots s'est réduit à un seul élément).

Les deux derniers exemples ne signifient point qu'il soit possible de munir le même langage de plusieurs sémantiques. Cela n'a lieu que pour les langages non formels (langues naturelles, par exemple).

Notons que, dans la troisième méthode, le langage joue un rôle secondaire, il représente une sorte de langage des numéros qui servent à énumérer les éléments de l'ensemble de mots qui correspond à un mot x_i du sous-langage Λ.

En ce qui concerne la quatrième méthode, on la retrouve dans les langages algorithmiques tels que langage des schémas logiques, ALGOL-60, etc. Selon cette méthode, on fait correspondre à tout mot un objet (une application constructive) qui représente, en général, un ensemble infini d'éléments.

Les explications données ici ne prétendent pas à la rigueur.

Il est caractéristique pour les quatre méthodes principales de définition de la sémantique qu'elles font correspondre à un mot du sous-langage Λ un objet au moyen d'un algorithme ; cette correspondance représente justement le sens complet ou partiel du mot.

La sémantique est dite *formelle* du fait que la correspondance entre les constructions du langage et leur « sens » s'établit au moyen d'un algorithme. Le sens d'une phrase est alors déterminé d'une manière univoque par sa structure, sa forme.

Il est démontré que, quel que soit le langage (engendré par une grammaire déductive ou par une grammaire inductive) sur lequel (ou sur un sous-langage duquel) on définit une sémantique formelle, il existe un langage indépendant du contexte (normal) de même sémantique. Cette affirmation veut dire que, dans le but de transmission d'information, on aurait pu bien se borner aux langages indépendants du contexte. Mais du point de vue pratique, ces langages s'avèrent, dans certains cas concrets, moins commodes que les langages dépendant du contexte.

5.5.6. Remarques sur la traduction. Pour fixer les idées, plaçons-nous dans le cas d'une traduction par la méthode de compilation. Les langages de programmation qu'on emploie actuellement sont relativement simples, quant à leur structure. Leurs phrases sont les constructions les plus simples, les mots qui admettent la

décomposition en éléments encore plus simples (instructions, descriptions de l'information, signes de passage, signes de groupement des éléments). Les éléments mentionnés se distinguent par certains indices. Avec une liste d'indices on peut faire une analyse syntaxique du programme d'entrée. C'est justement cette analyse syntaxique des éléments principaux qui est à la base de beaucoup de compilateurs qu'on appelle maintenant « classiques ».

Outre le bloc d'analyse syntaxique d'après les indices, un compilateur « classique » comprend des blocs générateurs dont chacun transforme les éléments d'un certain type en groupes d'instructions de la machine, et un bloc d'édition qui s'occupe de la distribution de la mémoire pour le programme en composition, de l'attribution d'adresses réelles à ses instructions et de la mise au point définitive du programme. On prévoit en outre un bloc de contrôle syntaxique qui, avant la compilation, recherche les erreurs syntaxiques dont les types possibles sont déterminés soit à partir des considérations d'ordre général, soit par l'expérience.

Certains compilateurs récents, dits « syntaxiquement orientés » procèdent à l'analyse syntaxique par l'analyse grammaticale (et non d'après les indices des éléments). L'analyse grammaticale se réduit à appliquer les règles d'une grammaire générative déductive « en sens inverse ». Pour les langages de programmation ces règles sont des substitutions, et leur application « en sens inverse » consiste en ce qu'au lieu d'une substitution de la forme $P \rightarrow Q$ qui appartient à la grammaire on effectue la substitution $Q \rightarrow P$. On dégage ainsi dans le programme proposé, et ceci d'une manière unique (à condition que la grammaire satisfasse à certaines restrictions), des groupes de symboles correspondant aux symboles non terminaux de la grammaire qui sont les noms des types des éléments principaux du programme à compiler. En fin de compte, le compilateur trouve tous les éléments principaux qui seront transformés en groupes d'instructions par les blocs générateurs. Ensuite le bloc d'édition effectue la répartition de la mémoire pour le programme composé, attribue les adresses réelles à ses instructions et lui donne une forme définitive. Les règles de grammaire qu'on emploie pour l'analyse grammaticale sont habituellement mises sous la forme d'une table. Le contrôle syntaxique peut être réalisé par le bloc d'analyse syntaxique. Nous nous sommes bornés ici à l'analyse syntaxique descendante.

Tandis qu'un compilateur « classique » est strictement individuel, un compilateur syntaxiquement orienté peut être aisément adapté, en changeant sa table grammaticale et certains blocs générateurs, à la traduction d'un autre langage (pas trop « éloigné » du langage pour lequel il fut conçu).

CHAPITRE 6

PROGRAMMATION SYMBOLIQUE

§ 6.1. Notion d'autocode ou de langage de programmation symbolique (langage d'assemblage)

En abordant l'élaboration d'un système d'exploitation (§ 5.4), les chercheurs n'ont à leur disposition que le langage machine dont tous les instructions et l'algorithme d'exécution sont réalisés par les circuits de la machine. Pour rendre moins difficile l'emploi de ce langage, on recourt à la symbolisation dans la rédaction de programmes qui consiste à exprimer le code d'opération et les adresses par des symboles. Le langage correspondant est très simple. Puisque dans un tel langage, à toute instruction de machine il correspond l'unique opérateur, le programme qui assure la traduction des algorithmes donnés dans ce langage aura une forme bien simple dans le langage machine. Nous appellerons un tel langage, *langage absolu de programmation symbolique*, *ou autocode* 1 : 1 *pour le langage machine*, ou langage d'assemblage, et le programme de traduction sera appelé assembleur.

C'est au moyen du langage absolu de programmation symbolique que l'on construit le système d'exploitation du calculateur. Certaines restrictions étant imposées à la disposition du système d'exploitation dans la mémoire, l'assembleur qui passe sur le programme rédigé en langage absolu de programmation symbolique n'effectue pas la répartition automatique de la mémoire, il réalise seulement le remplacement des adresses symboliques par les adresses réelles, tout en estimant les prescriptions du programmeur.

L'assembleur est rédigé dans le langage machine. Le système d'exploitation modifie la structure du langage machine du point de vue de l'utilisateur, en formant le système exécutif.

Si l'on part du langage du système exécutif, on peut construire un langage de programmation symbolique qui diffère du langage absolu de programmation symbolique par le seul fait que certains opérateurs en sont éliminés et sont ajoutées les instructions d'appel du système d'exploitation dites macro-opérateurs.

L'emploi du système d'exploitation peut entraîner des modifications du langage initial des opérandes. Par exemple, on en exclut les opérandes utilisés pour les interruptions (car, normalement, elles sont traitées par le système d'exploitation) et on y ajoute de

nouveaux opérandes, en général plus compliqués, que le système peut échanger avec les canaux de transmission.

Si à toute instruction et à tout opérande du langage du système exécutif correspond d'une manière univoque un opérateur et un opérande d'un langage de programmation symbolique, nous appellerons ce langage de programmation symbolique *autocode 1 : 1 pour le langage du système exécutif*.

Le perfectionnement ultérieur du langage de programmation symbolique consiste en ce que l'on charge l'assembleur non seulement de l'attribution des adresses réelles, mais aussi de la distribution de la mémoire. Pour les programmes composés après le système d'exploitation, les restrictions concernant la distribution de la mémoire sont moins sévères, bien que ces programmes puissent éventuellement faire partie du système du calculateur (par exemple, les compilateurs des langages orientés vers une classe de problèmes).

Notons que, si l'on conserve dans les opérateurs du langage symbolique la structure des instructions du système exécutif, la distribution automatique de la mémoire a un caractère restreint, car la structure même des instructions détermine déjà le type de la mémoire à réserver aux opérandes et la distribution automatique se fait à l'intérieur d'un type de mémoire.

Si, comme avant, à toute instruction et à tout opérande du langage du système exécutif correspond d'une manière univoque un opérateur et un opérande d'un langage de programmation symbolique, et si on n'établit au préalable aucune correspondance entre les adresses réelles et symboliques, alors on a affaire à un *autocode 1 : 1 pour le langage du système exécutif avec distribution automatique de la mémoire.*

L'étape suivante du perfectionnement des langages de programmation symbolique consiste en ce qu'on prévoit des mesures permettant de conserver toutes les opérations du langage machine d'une part et d'affaiblir les liens qui existent entre la structure des opérateurs et celle des instructions et de la mémoire d'autre part.

Il y a deux procédés bien usités qui permettent d'élargir les possibilités de construction des opérateurs et de simplifier la programmation.

Dans le premier procédé, la structure générale des opérateurs (le nombre d'adresses et leur nature) coïncide avec celle des instructions, mais il y a plus de possibilités de donner les adresses mêmes. L'adresse d'un opérande peut être écrite comme une expression composée d'adresses symboliques et de constantes reliées par des signes algébriques, par exemple. On obtient l'adresse réelle de l'opérande en calculant l'expression. Dans ce cas l'assembleur compose le groupe d'instructions nécessaires pour calculer l'expression proposée et le place avant l'instruction utilisant l'opérande ayant cette adresse.

On dit d'un langage qui admet des expressions pour les adresses des opérandes qu'il est un *autocode avec expressions d'adresse.*

Dans le second procédé, on conserve toutes les opérations « essentielles » qui se réalisent dans le cadre des instructions du système exécutif, mais on ne tient pas compte de la structure de la mémoire, et on élimine certaines opérations « non essentielles », à savoir les transferts entre les parties de la mémoire. Cela se fait dans le but de donner aux opérateurs du langage de programmation symbolique qui correspondent à des opérations « essentielles » les rangs déterminés par ces opérations, sous la condition que tous les opérandes soient accessibles en égale mesure (lorsqu'on s'adresse à la mémoire « principale »).

Dans le second procédé, l'assembleur complète automatiquement chaque opération essentielle par les opérations nécessaires d'échange entre les parties de la mémoire.

On dit alors qu'il y a un *autocode conservant les rangs des opérations essentielles.*

En réunissant les possibilités des deux derniers langages, on obtient un langage appelé *autocode avec expressions d'adresse conservant les rangs des opérations essentielles.* Par la suite ce langage sera considéré toujours à la place des langages dont il est réunion.

Nous dirons que le software d'un calculateur (d'un complexe de calculateurs) est complet par rapport au langage du système exécutif s'il comporte tous les langages particuliers de programmation symbolique énumérés, sauf le premier, ainsi que les assembleurs correspondants. Le premier langage est l'outil de travail des programmateurs du software, les utilisateurs n'en disposent pas.

Nous pouvons considérer les langages de programmation symbolique énumérés (sauf le premier) comme les sous-langages d'un langage complet de programmation symbolique (autocode complet).

Ainsi, le *langage complet de programmation symbolique* ou l'*autocode complet* représente l'ensemble des sous-langages (cas particuliers) suivants:

— l'autocode 1 : 1 pour le langage du système exécutif;

— l'autocode 1 : 1 pour le langage du système exécutif avec distribution automatique de la mémoire;

— l'autocode avec expressions d'adresse qui conserve les rangs des opérations essentielles.

Il est à remarquer que les softwares des calculateurs sont rarement munis d'un langage complet de programmation symbolique. On se contente d'ordinaire d'un seul langage particulier qui possède, dans une certaine mesure, les traits caractéristiques des langages énumérés.

Le trait commun à tous les langages de programmation symbolique est la conservation du langage des opérandes correspondant au langage du système exécutif et le fait qu'à chaque opération essen-

tielle du langage du système exécutif correspond un opérateur du langage de programmation symbolique.

Les paragraphes suivants du présent chapitre sont consacrés à la description des structures principales et des principes de construction d'un langage de programmation symbolique hypothétique, on y donne également des exemples de structures principales du langage de codage symbolique du calculateur « Minsk-32 » et du langage d'assemblage IBM-360.

§ 6.2. Alphabet d'un langage de programmation symbolique

La collection des symboles de base d'un langage de programmation symbolique, qui représente son alphabet, est habituellement formée par

1) les chiffres décimaux usuels;

2) les lettres latines ou latines et russes et le symbole ||*||;

3) les délimiteurs qui se divisent en signes d'opération et séparateurs. Les signes d'opération sont les signes d'opérations arithmétiques || + || — || × || / ||. Les séparateurs sont:

a) les symboles ||, ||. ||‘ ||’ ||$_{10}$|| ; ||⊔ || | || appelés respectivement virgule, point, guillemet gauche, guillemet droit, dix, point-virgule, blanc, trait vertical;

b) les parenthèses || (||) ||.

Dans les langages concrets de programmation symbolique, les alphabets diffèrent de celui qu'on vient de décrire, mais dans chacun on distingue à peu près les mêmes groupes.

Exemple 6.1. L'alphabet du langage de codage symbolique du calculateur « Minsk-32 » se compose de 79 symboles répartis en les groupes suivants:

1) les chiffres arabes ||0 ||1 ||2 || 3|| 4|| 5|| 6|| 7|| 8|| 9||;

2) les lettres:

a) les majuscules russes ||А||Б||В||Г||Д||Е||Ж||З||И ||Й||К||Л|| ||М||Н||О|| *) ||П||Р||С||Т||У||Ф||Х||Ц||Ч||Ш||Щ||Ы||Ь||Э||Ю||Я||;

b) les majuscules latines ||D||F||G||I||L||N||Q||R||S||V||W||Z||J||;

c) les symboles || × ||, || / || ↑ || = || ≠ || > || [||] || * || < ||;

3) les délimiteurs qui sont:

a) les signes d'opération ||+||—||;

b) les séparateurs || | || (trait vertical), ||⊔ ||(||)|| . ||‘ ||’ || ; ||$_{10}$|| : ||.

Exemple 6.2. L'alphabet du langage d'assemblage du système IBM-360 se compose de 256 symboles du code EBCDIC complétés du symbole || ||| (trait vertical). Une partie des symboles du code

*) Pour éviter la confusion de la lettre O et du zéro, on figure cette lettre comme Ø

EBCDIC est utilisée en tant que symboles du langage. Les constructions du langage d'assemblage sont formées à partir des symboles propres du langage. La possibilité d'utiliser tous les 256 symboles EBCDIC est autorisée par une mention spéciale.

Les symboles proprement dits du langage d'assemblage sont:

1) les chiffres décimaux arabes: ||0||1||2||3||4||5||6||7||8||9||;

2) les majuscules latines: ||A||B||C|| . . . ||Z||;

3) les symboles ||$||#||@||, ainsi que ||"||, ||&&||;

4) les délimiteurs:

a) les signes d'opération ||+||—||*||/|| (addition, soustraction, multiplication, partie entière du quotient des entiers);

b) les parenthèses ||(||)||;

c) les séparateurs ||.||=||'||,||&||␣||.

§ 6.3. Structures primaires du langage de programmation symbolique

On appelle *mot* une suite (chaîne) finie de symboles précédée et suivie d'un séparateur ||␣|| et n'ayant pas de symboles ||␣|| à l'intérieur.

Toutes les constructions primaires suivantes sont des cas particuliers des mots.

Nombres. Toute suite finie de chiffres s'appelle *entier sans signe.* On appelle *entier* tout entier sans signe ou tout entier sans signe précédé du signe ||+|| ou ||—||.

On appelle *partie fractionnaire d'un nombre* un entier sans signe précédé d'un point.

On appelle *exposant* une suite formée par un séparateur $||_{10}||$ suivi d'un entier (sans ou avec signe).

On appelle *nombre sans signe ni exposant* un entier sans signe ou une suite formée par un entier sans signe suivi de la partie fractionnaire d'un nombre.

On appelle *nombre sans signe et avec exposant* une suite formée par un nombre sans signe ni exposant suivi d'un exposant.

On appelle *nombre sans signe* un nombre sans signe ni exposant, ainsi qu'un nombre sans signe et avec exposant.

On appelle *nombre* ou bien un nombre sans signe, ou bien une suite formée par un signe ||+|| ou ||—|| suivi d'un nombre sans signe.

EXEMPLE 6.3.

a) ||00||0||01101||02830||12|| sont des entiers sans signe.

b) ||+0||—0||—01124||+1525||—100101|| sont des entiers.

c) ||.25||.00015||.0101||.9999|| sont des parties fractionnaires de nombres.

d) Les constructions $\|_{10}25\|_{10}-15\|_{10}+0102\|$ sont des exposants.

e) $\|0.25\|01101.0101\|12.9999\|8\|$ sont des nombres sans signe ni exposant.

f) $\|0.25_{10}\ 25\|38.21_{10}\ -1\|0.005_{10}\ +8\|25_{10}\ -10\|$ sont des nombres sans signe et avec exposant.

g) Les constructions e) et f) peuvent servir d'exemples de nombres sans signe.

h) Les constructions $\|+0.25_{10}\ 25\|25_{10}\ -10\|-0.01\|\ 0.15\|$ $\|0.7_{10}\ 3\|-0\|$ sont des nombres.

Les nombres du langage de programmation symbolique représentent une forme de notation des nombres au sens ordinaire du mot. On utilise le point à la place de la virgule. Le séparateur $\|_{10}\|$ permet d'employer la forme normale de notation des nombres.

On appelle *désignation symbolique d'un opérateur* une suite finie de lettres et chiffres que figure sur la liste de telles suites.

La désignation symbolique d'un opérateur correspond à une instruction ou à une suite d'instructions du calculateur auquel le langage de programmation symbolique est destiné. Les désignations symboliques d'opérateurs sont en général les codes mnémoniques des instructions.

La *désignation symbolique d'un dispositif* représente une suite finie de lettres et chiffres figurant sur la liste de telles suites.

Une désignation symbolique de dispositif correspond à un groupe de dispositifs du même type. Elle représente habituellement une abréviation mnémonique du nom des dispositifs du type donné.

Exemple 6.4. Une liste des désignations symboliques d'opérateurs est établie pour le langage de codage symbolique du calculateur « Minsk-32 ». Sans l'écrire, remarquons qu'elle contient 235 désignations symboliques dont chacune se compose de un à cinq symboles du langage de programmation symbolique. Pour certaines désignations symboliques d'opérateurs, le langage prévoit les désignations numériques équivalentes (elles représentent les codes de fonction des instructions correspondantes sous forme de nombres entiers octaux avec signe).

Les désignations symboliques des dispositifs forment une liste de 15 mots à deux lettres: ‖ МЛ ‖ bande magnétique, ‖ ЛС ‖ bande magnétique pour conserver les programmes du software, ‖ ВЛ ‖ lecteur de ruban perforé, ‖ ВК ‖ lecteur de cartes perforées, ‖ ЫЛ ‖ perforateur de ruban, ‖ ЫК ‖ perforateur de cartes, ‖ ПЧ ‖ imprimante, ‖ ММ ‖ machine à écrire du pupitre de commande, ‖ МБ ‖ tambour magnétique, ‖ МС ‖ processeur du calculateur « Minsk-32 », ‖ АБ ‖ dispositif de télécommunication (dispositif « Minsk-1560 »), ‖ ТГ ‖ dispositif pour liaisons télégraphiques (« Minsk-1550 »), ‖ ТФ ‖ dispositif pour liaisons téléphoniques (« Minsk-1571 »),

|| ЭК || écran de visualisation. Chacun de ces mots désigne un type de dispositif; dans une configuration concrète du calculateur, il peut y avoir plusieurs dispositifs de chaque type. Un dispositif particulier est désigné par la construction dans laquelle la désignation du type de dispositif est suivie du séparateur || , || suivi à son tour du numéro du dispositif donné soit par un nombre octal à deux chiffres, soit par un mot (adresse symbolique, voir exemple 6.6). Le nombre total des types de dispositifs qu'on peut brancher sur la machine « Minsk-32 » (sur le processeur) ne dépasse pas 32. Dans le langage, on peut désigner tous les types des périphériques par les nombres octaux à deux chiffres appartenant à l'intervalle de || 02 || à || 37 ||. Tous les dispositifs concrets sont désignés par les nombres octaux à trois chiffres de || 000 || à || 237 ||.

Exemple 6.5. Une liste des désignations symboliques d'opérateurs est établie dans le langage d'assemblage du système IMB-360. Les mots de la liste sont répartis en trois groupes : 1) les désignations des opérateurs correspondant aux instructions du système (160 mots environ) ; 2) les opérateurs commandant le fonctionnement de l'assembleur (40 mots environ) ; 3) les opérateurs correspondant aux macro-opérateurs du système d'exploitation (du système opératoire de IBM-360) — 20 mots environ.

On appelle *adresse symbolique élémentaire d'un opérande* une suite finie de symboles qui commence par une lettre.

On appelle *adresse symbolique élémentaire d'un opérateur* (*étiquette*) une suite finie de symboles qui commence par une lettre.

Bien que de même forme, les adresses symboliques élémentaires d'un opérande et d'un opérateur diffèrent par leurs significations.

Les adresses symboliques élémentaires d'opérandes sont les désignations des emplacements élémentaires de la mémoire (ou des suites d'emplacements élémentaires) dont les états représentent des opérandes.

Les adresses symboliques élémentaires d'opérateurs (étiquettes) sont les désignations ou bien des emplacements élémentaires de la mémoire dont les états représentent des opérateurs ou bien des suites d'emplacements élémentaires tels que l'état d'au moins l'un d'eux représente un opérateur, les états des autres étant des opérateurs ou des opérandes.

On appelle identificateur élémentaire une suite (*chaîne*) *finie de symboles qui commence par une lettre.* Les adresses symboliques élémentaires des opérandes et des opérateurs sont des identificateurs élémentaires.

A tout identificateur élémentaire correspond une adresse réelle exprimée par un entier sans signe. Le remplacement des identificateurs élémentaires par les valeurs numériques se fait au cours du

traitement de ces adresses par le programme de traduction approprié. Le nombre correspondant à un identificateur s'appelle sa valeur.

Eléments d'une chaîne. Une séquence non vide de symboles de base qui ne contient pas le symbole || | || (trait vertical) s'appelle *corps d'un élément de chaîne.* On appelle *élément de chaîne* le corps d'un élément encadré de symboles || | ||.

Lorsqu'une séquence de symboles contient plusieurs symboles || | || encadrant les corps d'éléments de chaîne, alors tout corps d'élément de chaîne considéré avec les symboles || | || qui l'encadrent représente un élément de chaîne. Ainsi, chaque symbole || | || interne de la séquence appartient à deux éléments voisins.

Un élément de chaîne représente la description d'une ligne de feuille de programmation. Le corps de cet élément représente le contenu d'une ligne de feuille de programmation.

§ 6.4. Adresses symboliques

En plus des adresses symboliques élémentaires (identificateurs élémentaires), on utilise dans le langage de programmation symbolique des constructions plus compliquées qui désignent des emplacements élémentaires de la mémoire ou des suites d'emplacements.

Les *arguments d'adresse* sont : l'identificateur élémentaire, l'entier sans signe, le symbole *.

Les *expressions d'adresse* sont : l'argument d'adresse, l'expression d'adresse entre parenthèses, deux expressions d'adresse liées par un signe d'opération.

Les *adresses symboliques* ou *identificateurs* sont : 1) l'expression d'adresse ne contenant pas le symbole || * ||; 2) l'expression d'adresse se composant d'un seul symbole || * ||; 3) l'expression d'adresse dans laquelle se suivent : un symbole || * ||; un signe d'opération; une expression d'adresse ne contenant pas le symbole ||*||; 4) deux expressions d'adresse sans symbole || * || liées par le symbole ||;||.

On détermine les valeurs des adresses symboliques : 1) en calculant la valeur d'une expression sans symbole || * || d'après les valeurs données des arguments d'adresse; 2) en affectant au symbole || * || une valeur égale à la valeur réelle de l'adresse de l'opérateur qui contient l'adresse symbolique exprimée par le symbole || * ||; 3) en calculant la valeur de l'expression d'adresse dans laquelle || * || prend la valeur déterminée comme indiqué au point précédent; 4) en additionnant l'entier sans signe rangé à l'emplacement élémentaire de la mémoire dont l'adresse est indiquée dans l'expression précédant le symbole ||;|| avec la valeur de l'expression qui suit ce symbole.

Le langage de programmation symbolique contient une construction qui détermine les valeurs et l'équivalence d'adresses symboli-

ques. La construction *égalité de deux adresses symboliques* se compose de trois éléments de chaîne. Le corps du premier élément est un identificateur élémentaire, le corps du deuxième, le mot || EGAL||, le corps du troisième, une expression d'adresse. Un identificateur élémentaire est dit numériquement défini, s'il est contenu dans le corps du premier élément de chaîne et que le troisième élément soit un nombre.

On appelle *identificateur élémentaire défini* ou bien un identificateur élémentaire numériquement défini, ou bien un identificateur élémentaire écrit dans le premier corps du premier élément d'une expression d'égalité dont le corps du troisième élément contient une expression dont tous les identificateurs élémentaires sont définis.

EXEMPLE 6.6. Dans le langage de codage symbolique, une adresse symbolique élémentaire se compose de un à cinq symboles alphanumériques, le premier devant être alphabétique. Tous les autres symboles sont défendus.

Les arguments d'adresse sont : les adresses symboliques élémentaires, les entiers positifs décimaux ou octaux, le symbole || * ||. Les nombres sont écrits sans signe, l'intervalle de représentation pour les nombres décimaux est $1 \leqslant \alpha \leqslant 65536$, celui des nombres octaux étant $1 \leqslant \beta \leqslant 177777$. Les nombres octaux sont suivis du symbole || B|| pour les distinguer des nombres décimaux.

Il y a quatre types d'expressions d'adresse.

Dans le premier, on admet l'utilisation des signes d'opération || + || et || — ||. Le premier argument d'adresse de l'expression est toujours une adresse symbolique élémentaire, les autres étant soit des adresses de même nature, soit des nombres. La quantité totale des symboles dans une expression ne dépasse pas 39.

Le deuxième type d'expression d'adresse représente une somme ou une différence de deux arguments d'adresse dont le premier est le symbole || * || et le deuxième, un nombre décimal ou octal.

Le symbole || * || représente l'adresse symbolique de l'emplacement élémentaire où est écrit l'opérateur contenant une expression de ce type.

Une expression d'adresse du troisième type se compose de deux arguments séparés par une virgule || , ||.

Le premier argument d'adresse est l'adresse de la cellule de mémoire contenant la base, le second étant une adresse relative. Les arguments peuvent être exprimés soit par un nombre (de 0 à 3 le premier et de 0 à 2047 le second), soit par une adresse symbolique élémentaire (dont la valeur se conserve lors des transferts du programme dans la mémoire). La valeur d'une expression de ce type (adresse réelle) s'obtient comme la somme de la base et du nombre ou du contenu de la cellule dont l'adresse symbolique élémentaire

est mentionnée en argument, selon la nature du second argument.

Une expression du quatrième type représente une adresse symbolique précédée du symbole || : || (deux points). Cette notation signifie qu'au cours de l'exécution de l'instruction à laquelle est associé l'opérateur contenant cette expression, il se forme l'adresse qui représente la somme de l'adresse réelle (correspondant à l'adresse symbolique mentionnée) et du contenu de la cellule servant de registre d'index indiquée dans le même opérateur (instruction).

Le langage de codage symbolique dispose de deux constructions permettant de définir les identificateurs élémentaires. La première se compose de trois éléments de chaîne. Le corps du premier élément contient cinq symboles dont les premiers sont des symboles de l'identificateur élémentaire (les derniers étant éventuellement des blancs si l'identificateur contient moins que cinq symboles). Le corps du deuxième élément contient le mot || ЗНАЧ ||, et le corps du troisième contient, dans l'ordre, un nombre (sous une forme adoptée pour les expressions d'adresse) ou une expression du troisième type, un symbole || ⊔ || et un mot quelconque formé de symboles du langage de programmation symbolique qu'on appelle commentaire. La longueur du corps du troisième élément est de 39 symboles. Si le commentaire se compose des seuls symboles || ⊔ ||, on dit qu'il est absent.

La seconde construction diffère de la première par le fait que le corps du deuxième élément contient le mot || ЭКВ ||, et celui du troisième élément contient (au lieu d'un nombre ou d'une expression d'adresse du troisième type) une expression du premier type.

Exemple 6.7. Dans le langage d'assemblage du système IBM-360, une adresse élémentaire symbolique est composée de un à huit symboles, le premier devant être alphabétique. Les symboles || $ ||, || # || et || @ || sont considérés comme lettres.

Les arguments d'adresse sont: les adresses symboliques élémentaires; les entiers (sans signe) décimaux, hexadécimaux ou binaires; les codes alphanumériques, le symbole || * ||. Chacun des arguments d'adresse énumérés (sauf l'adresse symbolique élémentaire) peut occuper au plus trois octets.

Un nombre décimal s'écrit sans signe, se compose de chiffres, aucun autre symbole n'étant admis (y compris le point et la virgule). Le nombre appartient à l'intervalle de 0 à 16777215, l'assembleur le convertit en forme binaire.

Un nombre hexadécimal s'écrit entre guillemets || ' ||, on le fait précéder du symbole || X ||. Le nombre se compose de un à six chiffres. Exemples: || X'A9C120' ||; || X'2' ||.

Un nombre binaire s'écrit entre guillemets, sa notation est précédée du symbole || B ||. Le nombre contient de un à 24 chiffres. Exemples: || B'101010010000010010100000' ||; || B'10' ||.

Un code alphanumérique s'écrit entre guillemets, sa notation est précédée du symbole || C ||. Il contient de un à trois symboles. Tous les symboles du code EBCDIC, sauf || ' || (guillemets) et || & ||, sont admis. Si, tout de même, l'un de ces deux symboles doit être inclus dans le code, on l'écrit deux fois de suite. Exemples : les notations || C'XA' ||; || C'9' 'A' ||; || C'&&' 'B' || définissent respectivement les codes alphanumériques XA, 9A, &B.

Le symbole || * || représente l'adresse symbolique élémentaire du premier octet de celui des opérateurs du langage d'assemblage qui a le symbole || * || pour argument d'adresse.

Une expression d'adresse est composée d'après les règles de l'algèbre élémentaire. On y utilise les arguments d'adresse, les signes des opérations algébriques et les parenthèses. Dans les différentes versions du software IBM-360 on admet respectivement : a) 16 arguments et cinq niveaux de parenthèses ; b) 8 arguments et trois niveaux de parenthèses ; c) trois arguments et un niveau de parenthèses. Une expression ne peut commencer par un signe d'opération, il est interdit d'écrire successivement deux signes d'opération ni deux arguments d'adresse. L'argument d'adresse || * || ne peut occuper que la première place dans une expression.

Grâce aux restrictions signalées, la fonction du symbole || * ||, employé dans ce langage pour noter la multiplication et aussi l'adresse symbolique élémentaire est interprétée correctement par l'assembleur.

Le résultat de calcul d'une expression ne doit pas dépasser $2^{24} - 1$, bien que les résultats intermédiaires puissent atteindre la valeur $2^{31} - 1$. La division se fait sans arrondissement. La partie entière du quotient est considérée comme son résultat. Le résultat d'une division par le zéro est nul.

Si la valeur calculée d'une expression dépend de l'implantation du programme dans la mémoire, l'expression est dite translatable. Elle est dite absolue si sa valeur est invariante par déplacement.

Le langage d'assemblage contient une construction de définition des identificateurs élémentaires. Elle se compose de sept éléments de chaîne successifs. Le corps du premier contient toujours huit symboles dont les premiers sont tous les symboles de l'identificateur élémentaire concret, les derniers pouvant éventuellement être des blancs || ␣ || si l'identificateur en question contient moins que huit symboles. Les corps des deuxième, quatrième et sixième éléments contiennent le symbole || ␣ ||, le corps du troisième élément contient le mot || EQU ␣ ␣ ||, celui du cinquième représente soit une adresse symbolique complétée par les symboles || ␣ || jusqu'à 55 (le total des symboles dans le corps du cinquième élément) soit une adresse symbolique et un mot arbitraire du langage d'assemblage, séparés par un blanc. Ce mot arbitraire s'appelle commentaire. Le corps du septième élément se compose de huit blancs.

§ 6.5. Langage des opérandes lié au langage de programmation symbolique

L'alphabet du langage des opérandes coïncide avec celui du langage de programmation symbolique.

Les structures primaires du langage de programmation symbolique telles que le mot, le nombre et l'identificateur élémentaire sont des structures primaires du langage des opérandes.

Les opérandes des algorithmes donnés dans le langage de programmation symbolique représentent ce qu'on appelle états de mémoire. On appelle *état de mémoire* une notation se composant de deux parties: état de mémoire extérieure et état de mémoire intérieure.

Un *état de mémoire intérieure* réunit les *états des emplacements élémentaires de mémoire* ou des *champs de mémoire*. A tout champ de mémoire on fait correspondre une adresse exprimée par un entier sans signe.

Un *état de mémoire extérieure* réunit lui aussi les états des emplacements élémentaires (des champs de mémoire) auxquels correspondent leurs adresses dans le langage machine. Dans le langage du système opératoire, un état de mémoire extérieure est déterminé par l'ensemble des états des zones de mémoire extérieure.

On appelle *zone de mémoire extérieure* (*zone extérieure*) une suite d'états des emplacements élémentaires de mémoire extérieure, composée de trois éléments : le corps du premier contient un identificateur de la zone, celui du deuxième, un entier décimal sans signe qui indique le nombre d'emplacements élémentaires de mémoire extérieure dans la zone, celui du troisième, les états de ces emplacements.

A toute zone de mémoire extérieure on peut faire correspondre une zone de mémoire intérieure.

Une *zone de mémoire intérieure* (*zone intérieure*) représente une construction qu'on obtient en remplaçant dans la définition d'une zone extérieure les emplacements de mémoire extérieure par ceux de mémoire intérieure.

On appelle *zone* aussi bien une zone extérieure qu'intérieure.

L'état de mémoire intérieure ou extérieure est donné de l'une des deux manières: soit au moyen du langage des opérandes, soit au cours de la traduction des opérateurs en langage machine. Comme l'état d'un emplacement élémentaire de mémoire peut être opérande d'opérateurs du langage, nous distinguons les opérandes-objets et les opérandes-instructions.

Un *opérande-instruction* représente l'état de mémoire qui joue le double rôle au cours de l'exécution du programme: celui d'une instruction correspondant à un opérateur du langage et aussi d'un opérande utilisé dans un opérateur du langage.

Un *opérande-objet* est un état d'emplacement élémentaire de mémoire qui ne figure qu'en tant qu'opérande d'un opérateur du langage.

On distingue les opérandes-objets nommés et les opérandes directs.

On appelle *opérande nommé* l'état d'un emplacement de mémoire auquel est affectée une adresse symbolique. Les opérandes nommés sont les emplacements élémentaires de mémoire intérieure et les zones intérieures ou extérieures.

On appelle *opérande direct* ou bien un opérande rangé dans un emplacement élémentaire de mémoire dont l'état représente une instruction correspondant à l'opérateur utilisant cet opérande, ou bien un opérande identiquement constant.

On ne considère dans le langage des opérandes que les opérandes-objets nommés.

Convenons que le langage de programmation symbolique utilise pour le rangement des opérandes des champs de longueur variable de mémoire intérieure. Bornons les longueurs maximales des champs à N_1 pour les mots, N_2 pour les nombres décimaux, N_3 pour les nombres octaux et N_4 pour les nombres binaires. Les nombres octaux et binaires sont des cas particuliers d'entiers sans signe, on les note au moyen des chiffres || 0|| 1|| 2|| 3|| 4|| 5|| 6|| 7|| et || 0|| 1|| respectivement.

La construction du langage qui décrit l'état d'un emplacement élémentaire de mémoire intérieure représente une suite de trois éléments de chaîne. Le corps du premier élément contient l'adresse symbolique élémentaire d'un opérande, le corps du deuxième contient le mot || AFFECTER||, le corps du troisième est l'une des quatre constructions dont les éléments sont respectivement :

1) un symbole || C||, deux symboles || ( )||, un mot contenant de un à N_1 symboles arbitraires de l'alphabet du langage, deux symboles || ( )||;

2) un symbole || Ч||, deux symboles || ( )||, un nombre décimal ayant de un à N_2 chiffres, deux symboles || ( )||;

3) un symbole || B||, deux symboles || ( )||, un nombre octal contenant de un à N_3 chiffres, deux symboles || ( )||;

4) un symbole || Д||, deux symboles || ( )||, un nombre binaire contenant de un à N_4 chiffres, deux symboles || ( )||.

Exemple 6.8. Dans le langage de codage symbolique pour le calculateur « Minsk-32 », il est possible de donner l'état d'une cellule (d'un emplacement élémentaire de mémoire) correspondant à un opérande-objet nommé isolé de l'une des manières suivantes :

a) Nombres décimaux en virgule flottante :

— avec un point qui sépare les parties entière et fractionnaire (point décimal) et avec un signe (|| +35.86|| —283.15|||| +83.|| ||—.00967||);

— avec un exposant qu'on exprime par une puissance de dix et que l'on note en utilisant le symbole $\|_{10}\|$ ($\| +0.027_{10}2 \|$ $\| -2._{10} - 7 \|$ $+284_{10}2 \|$).

Un nombre décimal en virgule flottante détermine l'état d'une cellule de mémoire rapide (étant automatiquement converti en un nombre binaire en virgule flottante).

b) Nombres décimaux en virgule fixe:

— ils sont donnés de la même manière que les nombres décimaux en virgule flottante, mais l'on fait suivre chacun d'eux du symbole || Ф|| (|| 0.62Ф|| $-0.03_{10} - 1$Ф|| $+0.9_{10} - 3$Ф||).

Un nombre décimal en virgule fixe est soit automatiquement converti en binaire et l'état de la cellule s'exprime alors par un nombre binaire en virgule fixe, soit représenté en décimal codé binaire en virgule fixe. Dans ce cas on écrit la lettre || Д || à la suite du nombre et de la lettre || Ф|| (|| +0.3ФД — $0.08_{10} - 2$ФД||).

c) Nombres entiers décimaux en virgule fixe multipliés par 2^{-36}:

— ils sont écrits comme des entiers signés (|| +41|| —837|| — —18||).

Ils sont soit automatiquement convertis en binaire, soit représentés en décimal codé binaire; dans le dernier cas on fait suivre le nombre du symbole || Д|| (|| +43Д|| —1024Д|| +18Д||).

d) Nombres entiers octaux en virgule fixe multipliés par 2^{-36}:

— ils s'écrivent avec signe, chaque nombre est suivi de la lettre || B|| (|| —7134B|| +2B|| —100B||).

e) Textes:

— ils se composent de un à cinq symboles quelconques de l'alphabet du langage des opérandes (le même alphabet que celui du langage de codage symbolique, voir l'exemple 6.1); si un texte contient moins que cinq symboles, on ajoute autant de blancs || ⊔ || qu'il y manque de symboles. En mémoire de la machine les symboles du texte, y compris le blanc, sont représentés en code binaire.

f) Adresse d'une cellule — registre d'index:

— elle est donnée soit par une adresse symbolique élémentaire, soit par une expression du premier type (voir l'exemple 6.6), soit par un entier positif (on omet alors d'écrire le signe || +||) ou négatif, décimal ou octal, compris dans l'intervalle de 0 à $2^{16} - 1$; un nombre octal est suivi du symbole || B||; la notation est interprétée comme l'état des seize positions de la cellule réservées à une adresse, les nombres décimaux sont convertis en binaire; si la valeur de l'opérande est négative, on prend pour l'état des seize positions mentionnées le complément à 2^{16}; si l'adresse est donnée par une expression, alors les seize positions mentionnées prennent l'état des seize positions d'adresse de la cellule dont l'adresse réelle est égale à la valeur de l'expression, cet état ne dépendant pas des déplacements du programme.

g) Adresse d'une cellule:

— elle est donnée soit par une adresse symbolique élémentaire, soit par une expression du premier type (voir l'exemple 6.6), soit enfin par un nombre entier positif, décimal ou octal (noté d'après les règles du point précédent). L'opérande est l'état des seize positions d'adresse de la cellule ; cet état est égal à la valeur de l'expression ou du nombre donné et varie (dans le cas d'une expression) si l'on déplace le programme.

Dans le langage de codage symbolique sont prévues les constructions suivantes pour définir l'état d'emplacements élémentaires de mémoire (de cellules) :

— définition des nombres :

la construction se compose de trois éléments de chaîne, le corps du premier contient l'adresse symbolique élémentaire d'un opérande, le corps du deuxième, le mot || КЧ ⊔ ⊔ ⊔ ||, le corps du troisième, un nombre sous l'une des formes a), b), c), d) et un texte (commentaire) qui le suit et qui en est séparé par un espace blanc ; le corps du troisième élément contient 39 symboles. Un commentaire est considéré comme vide s'il ne contient que les symboles || ⊔ || ;

— définition du texte :

cette construction diffère de la précédente par le fait que le corps du deuxième élément de chaîne contient le mot || КТ ⊔ ⊔ ⊔ || et celui du troisième, un mot écrit d'après les règles du p. e) au lieu d'un nombre ;

— définition du contenu d'une cellule sous forme d'index :

cette construction diffère de la construction « définition des nombres » par le fait que le corps du deuxième élément de chaîne contient le mot || КИ ⊔ ⊔ ⊔ || et celui du troisième, un mot écrit d'après les règles du p. f) au lieu d'un nombre ;

— définition du contenu d'une cellule sous forme d'adresses :

cette construction diffère de la construction « détermination des nombres » par le fait que le corps du deuxième élément de chaîne contient le mot || КА ⊔ ⊔ ⊔ || et celui du troisième se compose de deux notations faites conformément aux règles du p. g) et séparées par le symbole || ; ||. Les valeurs des notations dans le corps du troisième élément déterminent deux nombres entiers occupant la première et la troisième adresses d'une cellule dont tous les autres chiffres sont nuls.

Exemple 6.9. Nous avons déjà dit (voir l'exemple 2.29) que le système IBM-360 dispose de plusieurs méthodes de division de la mémoire rapide en emplacements élémentaires (cellules). Toutes ces méthodes sont basées sur la division de toute la mémoire en octets, i.e. cellules contenant huit positions binaires chacune. Les octets ont pour adresses les numéros successifs de 0 à 1677216. (La capacité de la mémoire varie selon la version, si elle est inférieure au nombre indiqué, les numéros inexistants ne sont pas utilisés).

Un octet est l'unité de mesure du volume de mémoire, et c'est en cas particulier le contenu d'un octet représente un opérande.

Notons préalablement que les symboles || A || B || C || D || E || F || du langage d'assemblage apparaissent dans les constructions de deux classes: dans les nombres hexadécimaux (entiers sans signe) où ils représentent respectivement les chiffres dix, onze, douze, treize, quatorze et quinze, et dans les mots où ils sont des lettres au sens ordinaire du mot.

Il y a deux types d'opérandes dans le système IBM-360: de longueur fixe et de longueur variable.

Un opérande de longueur fixe à la longueur de deux, quatre ou huit octets. Les adresses des cellules respectives sont des multiples de deux, quatre ou huit. Les opérandes correspondants s'appellent « demi-mot », « mot » et « double mot ».

Un opérande de longueur variable représente l'état d'une suite d'octets (d'un champ). Il y a deux types de tels champs. Les champs du premier type sont d'une longueur de un à 256 octets chacun, ceux du deuxième type contiennent de deux à 16 octets.

Notation des opérandes de longueur fixe. Ils sont notés de deux façons:

a) Comme nombres décimaux convertis en nombres binaires en virgule fixe:

— on écrit un nombre décimal entre deux symboles || ' ||. Cette notation est précédée du symbole || H || ou || F ||; le nombre décimal est converti en un nombre binaire qui est l'état ou bien d'un demi-mot (si l'on avait employé le symbole || H ||), ou bien d'un mot (pour le symbole || F ||); si le nombre décimal est noté avec une partie fractionnaire séparée par un point, cette dernière est ignorée;

— on peut mettre, entre le symbole || H || ou || F || et le nombre décimal en guillemets, deux facteurs (ou l'un d'eux), s et w, précédés des symboles respectifs || S || et || E ||; s indique que le nombre binaire obtenu doit être multiplié par 2^s, et w signifie que le nombre décimal de départ sera multiplié par 10^w (avant la conversion en binaire); s et w sont exprimés par des nombres entiers, leurs plages de variation sont $-187 \leqslant s \leqslant +346$; $-85 \leqslant w \leqslant +75$; si une construction commence par le symbole || S || ou || E ||, on convertit en binaire la partie entière et la partie fractionnaire du nombre, s'il est noté avec un point décimal.

Dans le système IBM-360, les nombres binaires en virgule fixe qui occupent un demi-mot ou un mot ont la virgule placée à droite (sont des entiers) et le signe dans la position de rang le plus élevé (à gauche). Les nombres négatifs sont représentés dans le complémentaire, donc, pour donner un nombre négatif décimal, il faut écrire entre guillemets son complément à 10^5 (avec le symbole || H ||) ou à 10^9 (avec le symbole || F ||), ou à une puissance de 10 plus élevée.

Par exemple, les nombres décimaux || H'37' || H'37.25' || F'16777215' ||, || E'99966' || (ce dernier représente la notation du nombre négatif décimal —34), || HS2'37.25' || HE1'3.725' || ||FS'—1116777215'|| sont exprimés dans le système héxadécimal par les demi-mots ou mots suivants: || 0025 || 0025 || 00FFFFFF || FFDF || 0895 || 0895 || 00001FFF || respectivement.

b) Comme nombres décimaux convertis en nombres binaires en virgule flottante:

— un nombre décimal avec signe contenant une partie entière et une partie fractionnaire séparées par un point (ou un nombre entier sans point) et noté entre deux symboles || ' || est converti en binaire et est écrit dans une cellule de 32 positions binaires (un mot), si la notation est précédée du symbole || E ||, ou dans une cellule de 64 positions binaires (double mot), si la notation est précédée du symbole || D ||; entre le symbole || E || (ou || D ||) et la notation entre guillemets on peut mettre une construction dans laquelle le symbole || E || est suivi d'un nombre entier w qui indique la multiplication par 10^w du nombre décimal proposé (avant sa conversion), ou une construction se composant du symbole || S || et d'un nombre entier s (où $0 \leqslant s \leqslant 14$) qui provoque le décalage de la mantisse du nombre binaire obtenu vers la droite de $s \times 4$ positions binaires (ce qui est équivalent à la multiplication par 16^{-s}).

Dans le système IBM-360 les nombres binaires en virgule flottante se représentent de deux façons: par un mot ou par un double mot. Dans les deux cas la virgule est placée devant le chiffre le plus significatif de la mantisse, le signe occupe la position extrême gauche (nulle) du code, le signe plus est figuré par 0, et le signe moins, par 1. Les sept positions suivantes (de la 1-e à la 7-e) sont réservées à la caractéristique qui représente l'exposant du nombre augmenté de +64. C'est l'exposant d'une puissance de 16, il peut varier de —64 à +63 (ce qui correspond à l'intervalle des nombres décimaux de 10^{-78} à 10^{75}). Les 24 (56) positions suivantes (de la 8-e à la 31-e ou de la 8-e à la 63-e) selon qu'il s'agit d'un mot ou d'un double mot contiennent la mantisse du nombre qui est normalisée si son chiffre de rang le plus élevé dans la base seize n'est pas nul. Les nombres non normalisés apparaissent lorsque $s \neq 0$.

Par exemple, les nombres décimaux || E'37' || D' — 37.25' || D'16777215' || ES2'37.25' || EE1'3.750' || E' — 0.75' || sont exprimés en notation hexadécimale virgule flottante comme || 22250000 || 6225400000000000 || 26FFFFFF00000000 || 2200540 || 22258000 || 40C00000 || respectivement.

Notation des opérandes de longueur variable. Les états d'un champ du premier type qui occupe de un à 256 octets expriment des opérandes dits logiques. Un opérande logique peut être donné sous l'une des trois formes:

a) Textes:

— ils se composent de symboles quelconques de l'alphabet du langage des opérandes, chaque mot est mis entre guillemets et cette notation est précédée du symbole || C || (|| C'ABCDEF' || C'24A' || C'END' ||) ; lorsqu'il faut inclure un symbole || ' || ou || & || dans un mot, on le répète deux fois : || '' || ou || && || ; ce symbole double sera considéré toujours comme un seul symbole (|| C'A''B' || C'B&& C' ||) ; lorsqu'un mot représente un texte composé de plusieurs mots séparés par des espaces blancs, ces blancs figurent également dans la notation :

(|| C'END ␣ OR ␣ UPDATE || C'LABEL TYPE' ||),

chaque blanc est compté comme symbole ;

— pour indiquer la longueur d'un texte, on peut mettre après le symbole || C || la lettre || L || et un nombre n qui donne la quantité de symboles dans le texte ; si le texte entre guillemets contient plus que n symboles, seuls les n premiers symboles sont perçus par la machine ; si le texte proposé contient moins que n symboles, alors ils sont tous perçus et ceux qui manquent sont considérés comme des blancs (|| CL5'MOUNT' || — tous les symboles sont perçus ; || CL3'ABCDE' || — sont perçus les symboles || ABC || ; || CL18'NOT' || sont perçus les symboles || NOT || et 15 blancs.

b) Codes hexadécimaux :

— leur notation ne diffère de celle des textes que par ceci : 1) à la place du symbole || C || qui précède un texte on met le symbole || X || ; 2) seules sont utilisées les lettres de l'alphabet qui désignent des chiffres hexadécimaux ; 3) le nombre n indique la quantité des couples de chiffres hexadécimaux qui sont à percevoir ; s'il dépasse le nombre de couples de symboles entre guillemets, alors le code sera automatiquement complété à gauche par des zéros (ainsi, le code || XL4'C1129' || sera considéré comme le code hexadécimal || 000C1129 ||) ; 4) si n n'est pas indiqué, alors un nombre entier d'octets sera rempli par les chiffres donnés ; dans le cas où le nombre de ces chiffres est impair, un zéro sera ajouté à gauche.

c) Codes binaires :

— ils diffèrent des codes hexadécimaux par ceci : 1) à la place du symbole || X || on met le symbole || B || ; 2) le nombre de symboles binaires || 0 || ou || 1 || à noter est un multiple de quatre ; 3) si à tout quaterne de symboles binaires on fait correspondre un nombre hexadécimal, alors la notation et la perception du code hexadécimal ainsi obtenu sont soumises aux règles 3) et 4) de l'alinéa précédent.

Un opérande logique est exprimé par les états de cellules consécutives de 8 positions binaires. A chaque symbole il correspond l'état d'une cellule exprimé par un code binaire déterminé (conformément à la table de correspondance du code EBCDIC).

Les opérandes de longueur variable du deuxième type sont des entiers décimaux codés binaires (avec la virgule à droite). Les nom-

bres décimaux en virgule fixe peuvent être donnés de deux manières :

a) Sous forme d'un nombre décimal dont chaque chiffre est représenté par l'état d'une cellule de 8 positions binaires :

— le nombre s'écrit entre guillemets en convention des entiers avec signe, la notation est précédée du symbole || Z || ; entre ce symbole et la notation en guillemets on peut mettre *n*, le nombre des chiffres à noter en tant qu'opérande ; si *n* dépasse la quantité des chiffres écrits en guillemets, on ajoute à gauche autant de zéros qu'il manque de chiffres ; si *n* est inférieur à la quantité des chiffres en guillemets, alors les chiffres les plus significatifs ne sont pas perçus ; un nombre décimal peut contenir jusqu'à 15 chiffres et, compte tenu du signe, occuper jusqu'à 16 octets ; le signe est représenté par un octet (au signe plus il correspond le code || FC ||, au signe moins, le code || FD ||) ; le point décimal d'une notation est ignoré, tous les chiffres étant considérés comme se rapportant à la partie entière du nombre.

b) Sous forme d'un nombre décimal dont chaque couple de chiffres est représenté par l'état d'une cellule de 8 positions binaires :

— les règles de notation des nombres décimaux de ce type diffèrent des règles du point précédent par ce que : 1) devant la notation en guillemets on met le symbole || P || ; 2) dans une cellule on écrit deux chiffres codés binaires ; 3) le nombre *n* indique la quantité de couples de nombres ; 4) un nombre peut contenir 31 chiffres, au signe plus il correspond le code || C ||, au signe moins, le code || D ||.

Signalons que dans certaines versions du langage d'assemblage pour les opérandes qui commencent par les symboles || H || F || E || || D || Z || P || B ||, le nombre *n* précédé des symboles || L || représente encore un indicateur de longueur et fournit la quantité de positions binaires (et non pas d'octets comme en cas d'un || L || sans point) à occuper par le code ou le nombre entre guillemets. L'enregistrement s'effectue alors à partir des poids forts du premier octet. S'il reste des positions non occupées du dernier octet, on y écrit des zéros.

Il existe des versions du software IBM-360 où il est permis d'utiliser en tant que *n*, *w* et *s* les résultats de calcul des expressions données sous la même forme que pour la notation des adresses symboliques (voir l'exemple 6.7).

En plus des opérandes déjà décrits, on définit, dans le langage des opérandes employé par le langage d'assemblage, les opérandes-objets isolés pour exprimer la valeur d'une adresse d'instruction. On lui réserve de un à quatre octets. Il y a quatre formes de représentation des opérandes-adresses :

a) La première représente une expression d'adresse (voir l'exemple 6.7) précédée du symbole || A || ; entre ce symbole et l'expression on peut mettre le symbole || L || et un nombre *n*, indiquant le nombre

d'octets (de 1 à 4) utilisés pour l'enregistrement du nombre binaire entier qui représente la valeur de l'expression ; si cette indication est absente, le nombre est rangé en quatre octets, les adresses du premier et du dernier octet étant multiples de quatre (un demi-mot) ; l'expression peut commencer par le symbole || * ||, alors, en tant que valeur de l'argument d'adresse désigné par ce symbole, on utilise le numéro du premier des octets gardant l'opérande-adresse ; l'expression d'adresse peut être absolue ou translatable (dans le dernier cas la longueur de l'opérande-adresse fait au moins trois octets).

b) La deuxième forme de notation d'opérandes-adresses diffère de la première par ce que, premièrement, on écrit le symbole || V || à la place de || A || et, deuxièmement, l'expression doit être translatable.

c) La troisième forme de notation d'opérandes-adresses diffère de la première par ce que le symbole || A || y est remplacé par || Y || et la longueur ne dépasse pas deux octets (si la construction avec le symbole || L || est absente, l'opérande-adresse est rangé à partir de la frontière du demi-mot).

d) Dans la quatrième forme de notation d'opérandes-adresses on distingue deux éléments : l'adresse d'une cellule contenant une base et une quantité interprétée habituellement comme l'adresse relativement à la base. Chacun de ces éléments est représenté par une expression, absolue ou translatable. On met l'expression désignant l'adresse de la base entre parenthèses et on la fait précéder de l'expression donnant l'adresse relative ; on met la notation ainsi obtenue entre parenthèses et la fait précéder du symbole || S ||.

Dans le langage des opérandes de l'assembleur IBM-360, il existe une construction de définition d'état d'emplacements élémentaires de mémoire (des champs de longueur variable). Cette construction représente une suite finie des notations dont chacune se compose de sept éléments de chaîne. Le corps du premier élément (dans la première notation) est formé par huit symboles dont les premiers sont tous les symboles de l'adresse symbolique élémentaire de l'opérande et les derniers sont des blancs || ⊔ || ; le corps du troisième élément contient le mot || DC ⊔ ⊔ ⊔ || ; le corps du cinquième contient l'une des constructions décrites dans le présent exemple, qui servent à noter les opérandes de longueur fixe ou variable et les opérandes-adresses. Si une construction exprimant un opérande de longueur variable contient plus que 54 symboles, elle se prolonge dans le corps du cinquième élément de la deuxième notation. On peut prolonger des constructions dans les cinquièmes éléments des notations suivantes. Les corps des deuxièmes et troisièmes éléments de toutes les notations sont constitués par les symboles || ⊔ ||. Le corps du sixième élément contient n'importe quel symbole de l'alphabet, sauf || ⊔ ||, si la notation n'est pas la dernière, ce corps contient le symbole || ⊔ || dans la dernière notation. Dans toutes les nota-

tions, sauf la première, les corps du premier et du troisième élément contiennent les mots || ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ || et || ⊔ ⊔ ⊔ ⊔ ⊔ || respectivement. Le corps du septième élément de chaque notation est le mot || ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ||.

La construction *définition de la zone intérieure* du langage des opérandes représente une suite de trois éléments de chaîne. Le corps du premier élément contient un identificateur élémentaire, celui du deuxième, le mot || ZONE ⊔ INTERIEURE ||, celui du troisième représente une suite des constructions de la forme 1 à 4 qu'on utilise dans la construction « définition d'état d'emplacement élémentaire de mémoire intérieure ».

La construction *définition de la zone extérieure* du langage des opérandes représente une suite de quatre éléments de chaîne. Le corps du premier élément contient la désignation symbolique d'un dispositif, celui du deuxième, un identificateur élémentaire, le corps du troisième, le mot || ZONE ⊔ EXTERIEURE ||, le corps du quatrième élément est la même construction que dans le corps du troisième élément de chaîne de la construction « définition de la zone intérieure ».

Exemple 6.10. Dans le langage de codage symbolique du calculateur « Minsk-32 », les quatres constructions suivantes sont prévues pour définir une zone intérieure :

1. Zone de nombres.

C'est une suite finie des constructions « définition de nombres » dont la première coïncide avec celle décrite plus haut (exemple 6.8), et les suivantes en diffèrent par le seul fait que le corps du premier élément de chaîne contient le mot || ⊔ ⊔ ⊔ ⊔ ⊔ ||.

2. Zone de textes.

Elle diffère de la construction « zone de nombres » par ce qu'à la place de la construction « définition de nombres » on y utilise la construction « définition du texte ».

3. Zone de cellules d'index.

Elle diffère de la construction « zone de nombres » par ce qu'à la place de la construction « définition de nombres » on y utilise la construction « définition du contenu d'une cellules sous forme du registre d'index ».

4. Zone de cellules d'adresse.

Elle diffère de la construction « zone de nombres » par ce qu'à la place de la construction « définition des nombres », on y utilise la construction « définition du contenu d'une cellule sous forme d'adresses ».

Exemple 6.11. Dans le langage d'assemblage du système IBM-360, la définition des zones intérieures se fait à l'aide de la même construc-

tion qui donne l'état d'emplacements élémentaires de mémoire, avec la seule différence que, dans les corps des cinquièmes éléments de chaîne, on écrit en guillemets la suite d'opérandes séparés par des virgules. Une telle notation est valable pour les constructions précédées de l'un des symboles ‖ Z‖ P‖ H‖ F‖ E‖ D ‖.

Notons que, dans le langage d'assemblage, on peut donner un facteur de multiplicité (de répétition) figuré par un entier décimal sans signe et non nul et qui s'écrit devant la construction commençant le corps du cinquième élément de la première notation de la construction donnant un emplacement élémentaire de mémoire (ou une zone). On met un facteur de multiplicité d au lieu de répéter d fois la première notation entre guillemets (pour les constructions précédées de l'un des symboles ‖ C‖ X‖ B‖), ou bien au lieu de répéter d fois les notations avec virgule (pour les constructions précédées de l'un des symboles ‖ Z‖ P‖ H‖ F‖ E‖ D‖).

Dans le cas particulier où $d = 0$, aucun opérande n'est introduit en mémoire; mais, lorsqu'on met $d = 0$ devant une construction où le premier guillemet est précédé d'un ‖F‖, sera assuré l'enregistrement du premier opérande donné dans la construction suivante définissant l'état de champ de mémoire (de zone) à partir du premier octet du mot le plus proche non occupé.

§ 6.6. Opérateurs du langage de programmation symbolique

6.6.1. Opérateurs d'action simples. Un opérateur de programmation symbolique représente une suite de chaînes dont chacune se compose de trois éléments. Le corps de chaque premier élément contient ou bien l'adresse symbolique élémentaire de l'opérateur, ou bien les symboles ‖ ⊔ ‖. Le corps de chaque deuxième élément contient ou bien la désignation symbolique de l'opérateur, ou bien les symboles ‖ ⊔ ‖. Le corps de chaque troisième élément représente une construction se composant d'adresses symboliques et d'opérandes directs (construction d'adresses).

Les opérateurs du langage se classent en trois catégories:

— opérateurs d'action;

— opérateurs de description des constructions d'opérateurs, opérateurs de description et d'allocation de mémoire;

— opérateurs de mise au point.

Parmi les opérateurs d'action il faut distinguer les opérateurs simples et les macro-opérateurs.

On partage les opérateurs simples en opérateurs arithmétiques (de traitement) et opérateurs de contrôle (de branchement).

Dans les opérateurs arithmétiques, toutes les adresses symboliques qui figurent dans les constructions d'adresses sont des adresses d'opérandes.

Voyons de plus près les constructions d'adresses d'opérateurs arithmétiques appelées indicateurs d'opérandes. Un indicateur d'opérandes représente une suite d'adresses symboliques d'opérandes et des opérandes directs séparés par les symboles || ( )||. Un exemple d'indicateur d'opérandes est fourni par la construction || AB + + 3 ( ) 17081 ( ) . . . ( ) . . . N||, où || AB + 3||, || 1708|| et || N || sont les adresses symboliques d'opérandes. Un autre exemple d'indicateur d'opérandes est la construction || AB + 3 ( ) = = +125 ( ) 1708 ( ) . . . ( ) N||, où || AB + 3||, || 1708|| et || N|| sont des adresses symboliques d'opérandes et la notation || = +125|| représente un opérande direct.

Le nombre d'adresses symboliques et d'opérandes directs dans un indicateur d'opérandes est déterminé aussi bien par le langage de programmation symbolique particulier que par les désignations symboliques d'opérateurs. Pour un même nombre d'adresses, les différentes désignations d'opérateurs peuvent conditionner l'utilisation différente des opérandes.

Par exemple, si un indicateur d'opérandes contient un seul élément et cet élément est une adresse symbolique, alors, selon la fonction de l'opérateur, cette adresse peut déterminer ou bien l'un des opérandes (donnée initiale ou résultat) d'une opération de rang un, le deuxième opérande étant situé dans l'additionneur de l'organe de calcul (emplacement élémentaire spécial de mémoire); ou bien l'un des opérandes d'une opération de rang deux (donnée initiale ou résultat, selon l'opérateur), deux autres opérandes étant représentés par l'état de l'additionneur avant et après l'exécution de l'opération.

Si un indicateur d'opérandes contient un seul élément qui est un opérande direct, alors cet opérande est toujours une donnée initiale.

S'il y a deux adresses symboliques dans un indicateur d'opérandes, alors elles représentent ou bien les deux opérandes d'une opération de rang un, ou bien deux opérandes d'une opération de rang deux (le troisième opérande est dans ce cas le contenu de l'additionneur). Si un indicateur d'opérandes contient trois adresses symboliques, elles déterminent tous les trois opérandes d'une opération de rang deux.

Lorsqu'un indicateur d'opérandes se compose de deux ou trois éléments, dont certains sont opérandes directs, ces derniers sont toujours des données initiales.

Les opérateurs arithmétiques déterminent l'exécution non seulement de l'opération principale (arithmétique, comparaison, etc.), mais aussi d'une opération qui l'accompagne et que l'on peut appeler calcul de « l'indice du résultat ». L'indice du résultat est rangé dans une cellule de mémoire spéciale (registre) qui n'a pas d'adresse numérique.

Les opérandes initiaux pour l'opération de calcul de l'indice du résultat sont les mêmes que pour l'opération principale de l'opérateur, et l'opérande-résultat est l'état du registre mentionné. L'indice prend certaines valeurs fixes, selon que sont satisfaites ou non les relations $\|\geqslant\|<\|=\|\neq\|$ entre des opérandes.

Dans un opérateur de contrôle, la construction d'adresses contient au plus deux adresses symboliques d'opérateurs (deux étiquettes). Dans cette construction on n'utilise pas les opérandes directs.

On distingue parmi les opérateurs de contrôle les opérateurs de branchement conditionnel et les opérateurs de branchement inconditionnel.

Les premiers assurent la réalisation de deux opérations: 1) calcul d'un prédicat et 2) passage de la commande à l'opérateur indiqué par une étiquette donnée. L'opération de calcul du prédicat utilise comme opérandes initiaux le contenu du registre d'indice du résultat et une constante déterminée par la nature de l'opérateur; elle représente la vérification de l'égalité de ces opérandes. Le résultat vaut l'unité si l'égalité a lieu, et zéro dans le cas contraire. La seconde opération utilise comme opérandes la valeur du prédicat et deux adresses symboliques, dont l'une est donnée dans la construction d'adresses, et l'autre ou bien est également donnée dans la construction d'adresses, ou bien représente l'« étiquette implicite » de l'opérateur immédiatement suivant l'opérateur considéré. L'étiquette implicite (l'adresse symbolique implicite) représente l'étiquette de l'opérateur logique considéré augmentée d'une unité (voir aussi § 6.7). Le résultat d'exécution de l'opérateur sera le contenu du champ de mémoire « étiquette courante » qui est la désignation de l'opérateur en cours d'exécution (§ 6.7).

La construction d'adresses d'un opérateur de branchement inconditionnel se compose d'une seule adresse symbolique.

Lors de l'exécution de l'opérateur de branchement inconditionnel l'adresse symbolique est transférée de la construction d'adresses dans le champ de mémoire « étiquette courante ».

EXEMPLE 6.12. Dans le langage de codage symbolique du calculateur « Minsk-32 », un opérateur d'action simple représente une construction se composant de trois éléments de chaîne. Le corps du premier contient soit l'adresse élémentaire symbolique d'un opérateur, soit le mot $\|\sqcup\sqcup\sqcup\sqcup\sqcup\|$. Le corps du deuxième élément contient la désignation symbolique de l'opérateur (voir l'exemple 6.4). Le corps du troisième élément est une construction d'adresses (indicateur d'opérandes). La construction d'adresses d'un opérateur arithmétique simple contient un ou deux éléments qui sont soit l'adresse symbolique d'un opérande, soit un opérande direct. Les règles de notation des adresses symboliques sont exposées dans l'exemple 6.6. Les règles de notation des opérandes directs-nombres

sont identiques aux règles de notation des nombres-opérandes nommés (exemple 6.8). Les règles de notation des opérandes directs-textes ne diffèrent des règles de l'exemple 6.8 pour les opérandes nommés-textes que par l'emploi de guillemets entre lesquels on met un tel opérande direct. Les opérandes directs sont appelés « littéraux ».

Deux éléments d'une construction d'adresses sont séparés par le symbole ||;||.

La manière dont on utilise un opérande donné soit par une adresse symbolique, soit directement est déterminée par la désignation symbolique de l'opérateur.

Remarquons (en précisant l'exemple 6.4) qu'il y a dans le langage une classe d'opérateurs réalisant des opérations de rang deux; les premières lettres de la désignation symbolique de ces opérateurs déterminent l'opération, la dernière, la manière d'utiliser les opérandes.

Si la dernière lettre est || 3||, alors les deux opérandes indiqués dans la construction d'adresses sont les données initiales de l'opération, et l'opérande-résultat est mis à la place du deuxième opérande (qui ne peut être donné comme opérande direct) et dans le registre de résultat (une cellule spéciale de mémoire).

Si la dernière lettre est || B||, alors les opérandes initiaux sont le contenu du registre de résultat et le premier opérande, l'opérande-résultat étant le contenu de la cellule indiquée par la deuxième adresse symbolique.

Si la dernière lettre est || P||, alors (bien que l'opération soit toujours de rang deux) la construction d'adresses contient un seul élément, et les opérandes-données initiales sont: le contenu du registre de résultat et ou bien le contenu de la cellule indiquée par la première adresse symbolique, ou bien un opérande direct. L'opérande-résultat s'inscrit dans le registre de résultat.

Si l'on n'ajoute aucun symbole, sauf || ⊔ ||, aux lettres désignant l'opération, alors les opérandes initiaux sont déterminés par le premier et le deuxième éléments de la construction d'adresses (adresses symboliques ou opérandes directs), le résultat étant contenu dans le registre de résultat.

En ce qui concerne les opérateurs décrits, il est à remarquer que dans ce langage, les expressions d'adresses du quatrième type (avec l'indication de la cellule d'index, voir l'exemple 6.6) sont écrites pour les deux adresses symboliques et non pour chaque adresse séparément. La construction d'adresses commence alors par le symbole ||:||, puis on met l'adresse symbolique de la cellule d'index, le séparateur ||;||, la première adresse symbolique, le séparateur ||;|| et la deuxième adresse symbolique.

Dans le langage de codage symbolique il y a des opérateurs pour les opérations de rang deux qui sont destinés à modifier le contenu des cellules utilisées comme registres d'index servant à leur tour à

modifier les adresses d'instructions. La construction d'adresses d'un tel opérateur commence par le symbole ||:|| suivi de l'adresse symbolique d'une cellule d'index, du séparateur ||;|| et de l'adresse symbolique du deuxième opérande (ou d'un opérande direct).

Il existe des opérateurs pour les opérations de rang un qui ont pour opérandes le contenu de cellules d'index. Selon la désignation symbolique de l'opérateur, sa construction d'adresses est celle que nous venons de décrire ou bien (pour les opérateurs utilisant les registres d'index et réalisant des opérations de rang un), ou bien se compose du symbole ||:|| suivi de l'adresse symbolique d'une cellule d'index. Dans le premier cas, l'un des opérandes est le contenu du registre d'index, l'autre, le contenu (ou une partie) de la cellule indiquée par l'adresse symbolique (ou l'opérande direct); dans le deuxième, les opérandes sont les contenus du registre d'index et du registre de résultat. Utiliser le contenu de telle ou telle cellule en tant que donnée initiale ou résultat dépend de la nature de l'opérateur.

Considérons encore un groupe d'opérateurs du langage qui réalisent des opérations de rang un. Les constructions d'adresses de ces opérateurs contiennent un seul élément représentant ou bien l'adresse symbolique d'une cellule, ou bien un opérande direct (si l'opérateur l'admet). L'un des opérandes est déterminé par la construction d'adresses, l'autre est le registre de résultat. La manière d'utiliser chaque opérande (donnée initiale ou résultat) est déterminée par la nature de l'opérateur.

Indiquons encore deux types d'opérateurs arithmétiques. Au premier se rapportent les opérateurs qui permettent d'utiliser comme opérande le contenu d'une partie de cellule de mémoire. Le deuxième type réunit les opérateurs dont les opérandes sont des états de cellules spéciales indicateurs.

La construction d'adresses d'un opérateur du premier type a deux éléments. Le deuxième élément est l'adresse symbolique d'une cellule, et le premier, un entier sans signe qui indique ce qu'on appelle « numéro de symbole », i.e. le numéro de l'un des groupes de positions binaires représentant le code binaire d'un symbole de l'alphabet du langage. Les opérandes sont : le contenu d'une partie de cellule qui est indiquée par la construction d'adresses, et une partie analogue du registre de résultat. La manière d'utiliser les opérandes est déterminée par la nature de l'opérateur.

L'opérateur de formation d'indicateurs a pour opérande-résultat l'état de cellules spéciales, dont le rôle est d'autoriser ou de défendre l'arrondissement ou la normalisation. La construction d'adresses de certains opérateurs de ce type contient des nombres entiers spéciaux représentant en binaire l'opérande-donnée initiale, pour d'autres, cette construction contient les désignations symboliques de cellules-indicateurs. Dans ce dernier cas plusieurs désignations sym-

boliques séparées par le signe || + || peuvent être présentes dans un opérateur.

Les opérateurs de branchement du langage de codage symbolique ont les constructions d'adresses de deux types : avec une ou deux adresses symboliques d'opérateurs. En tant qu'opérandes initiaux on utilise le contenu de tout le registre de résultat ou d'une partie de ce registre et une constante déterminée par la fonction de l'opérateur. Le calcul du prédicat consiste en la vérification de l'égalité des opérandes initiaux. Le résultat représente le contenu du champ de mémoire « étiquette courante », où est transférée, pour les opérateurs à deux adresses symboliques, l'une de ces adresses selon la valeur du prédicat. Pour les opérateurs dont la construction d'adresses a une adresse symbolique d'opérateur, le résultat, i.e. le contenu du champ « étiquette courante », est ou bien ladite adresse symbolique, ou bien l'étiquette implicite de l'opérateur suivant.

Notons que, dans le langage de codage symbolique, il y a des opérateurs du type mixte qui réalisent simultanément une opération arithmétique et une opération de formation du contenu du champ « étiquette courante » (formation de branchement). La construction d'adresses d'un tel opérateur se compose de deux adresses symboliques d'opérateurs dont la première est l'une des adresses symboliques où doit passer le contrôle (l'autre adresse de passage du contrôle est l'étiquette implicite de l'opérateur suivant), la deuxième indique la cellule renfermant l'opérande pour l'opération arithmétique de rang un. L'autre opérande est pris dans le registre de résultat. La manière d'utiliser les opérandes (comme résultat ou comme donnée initiale) est déterminée par la nature de l'opérateur.

Il se peut que la construction d'adresses d'un opérateur logique ou du type mixte contienne, en plus d'une ou de deux adresses symboliques, l'adresse d'une cellule d'index.

L'adresse symbolique de cette cellule est indiquée dans une expression (modifiée) du quatrième type (comme pour un opérateur arithmétique réalisant une opération de rang deux).

Exemple 6.13. Dans le langage d'assemblage du système IBM-360, un opérateur d'action simple représente une construction contenant sept éléments de chaîne (voir l'exemple 2.29, chapitre 2). Le corps du premier élément contient ou bien l'adresse symbolique élémentaire d'un opérateur, ou bien le mot || ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ||. Le corps du deuxième élément contient le symbole || ⊔ ||. Celui du troisième élément renferme la désignation symbolique de l'opérateur. Le corps du quatrième élément contient le mot || ⊔ ||, celui du cinquième, la construction d'adresses, celui du sixième, le symbole || ⊔ || et le corps du septième, le mot || ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ||.

Il existe dix types de constructions d'adresses des opérateurs arithmétiques. La désignation symbolique de l'opérateur détermine le type de la construction utilisée, le rang des opérations à exécuter et l'ordre d'utilisation des opérandes. La plupart des opérateurs arithmétiques déterminent l'exécution de deux opérations. Le résultat de l'une d'elles s'inscrit dans un emplacement de mémoire indiqué par l'adresse symbolique correspondante figurant dans la construction d'adresse. Les opérandes initiaux sont donnés eux aussi par des adresses symboliques. Le résultat de la seconde opération est représenté par l'état d'une cellule spéciale « indice de résultat », les opérandes initiaux de cette opération étant les mêmes que pour la première.

Une construction d'adresse du premier type contient deux adresses symboliques d'opérandes séparées par le symbole || , ||. Chacune de ces adresses détermine le contenu d'un registre de travail d'une capacité de quatre ou huit octets (un demi-mot ou un mot). Les opérandes sont interprétés, selon la nature de l'opérateur, ou bien comme des nombres en virgule fixe ou en virgule flottante, ou bien comme des codes binaires.

Une construction d'adresse du deuxième type contient une seule adresse symbolique désignant un registre. Cette construction est utilisée dans un seul opérateur qui réalise le transfert du contenu d'une partie déterminée (de la 2e à la 7e position) du registre (masque du programme) dans une cellule spéciale dont le contenu caractérise l'état du programme en exécution (mot courant du programme).

Une construction d'adresse du troisième type est formée par un opérande direct et ne s'utilise que dans un opérateur unique qui réalise le passage du programme-objet au superviseur. L'opérande direct (un octet) s'inscrit dans une partie d'une cellule spéciale qui caractérise l'état du programme interrompu et représente un « code d'interruption ».

Une construction d'adresse du quatrième type contient deux adresses symboliques séparées par le symbole || , ||. La première d'entre elles est l'adresse symbolique d'un registre. La deuxième est l'une des deux expressions spéciales. La première expression commence par une adresse symbolique de registre, puis on écrit successivement le symbole || (||, une adresse symbolique de registre, le séparateur || , ||, une adresse symbolique et, enfin, le séparateur ||) ||. L'expression décrite détermine l'ordre suivant de calcul d'une adresse réelle : les contenus des 24 positions les moins significatives des registres s'additionnent et la somme obtenue est ajoutée à la valeur de la dernière adresse symbolique de l'expression (cette valeur ne dépassant pas le nombre 4096).

L'expression de l'autre type représente la suite : une adresse symbolique, le séparateur || (||, l'adresse symbolique d'un registre, le séparateur ||) ||. En calculant cette expression on ajoute à la valeur

de la première adresse symbolique le nombre occupant les 24 positions les moins significatives du registre dont l'adresse est indiquée dans l'expression.

Une construction d'adresse du cinquième type représente une suite dont les premier et deuxième éléments sont des adresses symboliques de registres séparées par le symbole || , ||. Le deuxième élément est suivi d'un autre symbole || , ||, après quoi on met ou bien une adresse symbolique de mémoire rapide, ou bien une construction se composant d'une adresse symbolique de mémoire rapide (dont la valeur ne dépasse pas 4096), d'un symbole || (||, d'une adresse symbolique de registre, d'un symbole ||) ||. Dans le dernier cas, la valeur de l'adresse occupant la troisième place dans la construction d'adresse est calculée en additionnant le contenu des 24 positions du registre dont l'adresse est écrite entre parenthèses avec l'adresse symbolique précédant le symbole || (||.

Les opérateurs à construction d'adresse du cinquième type réalisent les transferts d'information entre les registres (à partir du registre indiqué à la première place de l'expression d'adresse jusqu'à celui indiqué à la deuxième place) et les cellules de mémoire principale (à partir de la cellule identifiée par le troisième élément de l'expression d'adresse) ou dans le sens inverse, entre la mémoire principale (3e élément de l'expression) et les registres (à partir de l'adresse indiquée à la première place jusqu'à celle indiquée à la troisième place de l'expression d'adresse). On compte les cellules modulo 2^{24}.

Une construction d'adresse du sixième type diffère de celle du cinquième type par ce qu'elle contient des adresses symboliques à la première et à la deuxième place seulement. La première place est occupée par l'adresse symbolique d'un registre, la deuxième, par une construction ayant la même forme que la construction qui occupe la troisième place dans l'expression d'adresse du cinquième type.

Les opérateurs à construction d'adresse du sixième type réalisent des décalages du contenu du registre dont l'adresse occupe la première place dans la construction d'adresse, d'un nombre de positions déterminé par les six positions les moins significatives de l'adresse symbolique indiquée à la deuxième place de la construction d'adresse. Le sens du décalage (vers la droite ou vers la gauche), son caractère (avec ou sans position du signe) et la longueur de l'opérande à décaler (32 ou 64 positions) sont déterminés par la nature de l'opérateur.

Une construction d'adresse du septième type représente une suite de deux éléments séparés par le symbole || , ||. Le premier élément s'écrit ou bien comme une expression d'adresse d'après les règles données dans l'exemple 6.7, ou bien se compose d'une adresse symbolique (dont la valeur ne dépasse pas 4096), d'un symbole || (||, d'une adresse symbolique et d'un symbole ||) ||. Le deuxième élément est une adresse symbolique.

Les opérateurs à structure d'adresse du septième type réalisent des opérations logiques de rang deux. Le premier opérande de telles opérations est le contenu de l'octet de mémoire rapide dont l'adresse est donnée par le premier élément de la construction d'adresse. Si cet élément s'écrit avec l'emploi de symboles || (|| et ||)||, la valeur de l'adresse symbolique s'obtient en additionnant l'adresse symbolique précédant le symbole || (|| et le contenu des 24 positions les moins significatives de la cellule dont l'adresse symbolique se trouve entre les symboles || (|| et ||)||.

Le second opérande (direct) est donné par les huit positions les moins significatives de l'adresse symbolique représentant le deuxième élément de la construction d'adresse. Le résultat s'inscrit à la place du premier opérande.

Une construction d'adresse du huitième type diffère de celle du septième type par ce qu'elle contient un seul élément qui est identique au premier élément de la construction du septième type.

Les opérateurs à construction du huitième type sont destinés à commander les entrées-sorties. L'adresse symbolique fournit les numéros d'un canal et d'un dispositif du canal.

Une construction d'adresse du neuvième type contient deux éléments séparés par le symbole || ,||. Chacun de ces éléments peut être mis sous l'une des deux formes suivantes. La première forme contient dans l'ordre: une adresse symbolique (dont la valeur ne dépasse pas 4096), un symbole || (||, une expression d'adresse (dont la valeur ne dépasse pas 256), un symbole || ,||, une adresse symbolique, un symbole ||)||. La deuxième forme diffère de la première par le fait de contenir entre les symboles || (|| et ||)|| une adresse symbolique simple.

Une construction d'adresse du dixième type se compose elle aussi de deux éléments séparés par le symbole || ,||. Le premier élément peut avoir l'une de deux formes indiquées pour les constructions du neuvième type. Le deuxième élément est ou bien une adresse symbolique, ou bien une construction formée par une adresse symbolique (à valeur ne dépassant pas 4096), un symbole || (||, une adresse symbolique, un symbole ||)||.

Les constructions d'adresse des neuvième et dixième types s'utilisent dans les opérateurs réalisant des opérations de rang deux sur les opérandes de longueur variable, décimales et logiques respectivement. La longueur des opérandes initiaux est déterminée par le premier terme de la suite se trouvant entre les symboles || (|| et ||)||.

Les adresses des opérandes sont soit indiquées par l'adresse symbolique, soit calculées en additionnant le contenu des 24 positions les moins significatives de la cellule dont l'adresse est indiquée à la deuxième place entre parenthèses, avec l'adresse symbolique précédant le symbole || (||. Le résultat de l'opération s'écrit à la place du premier opérande.

Pour tous les opérateurs décrits, l'exécution de l'opération principale est accompagnée de la formation de l'indice de résultat ce qu'on obtient en vérifiant une relation déterminée entre le résultat et une constante (dépendant de la nature de l'opérateur). La valeur de l'indice de résultat est représentée par l'état d'un registre spécial à quatre positions qui n'a pas d'adresse numérique. Les unités dans les positions respectives du registre témoignent que la relation en question est remplie, le zéro apparaît dans le cas contraire.

Il existe dans le langage d'assemblage environ trente opérateurs de contrôle qui réalisent des branchements conditionnels et inconditionnels et qui diffèrent par leurs désignations symboliques.

Pour ces opérateurs, on utilise les constructions d'adresses des premier, quatrième, cinquième et deuxième types décrits plus haut relativement aux opérateurs arithmétiques ainsi qu'une construction d'adresse spéciale.

Pour la construction d'adresse du premier type: a) la première adresse symbolique indique le numéro du registre où est inscrite la valeur de l'« étiquette courante » (l'adresse d'instruction), et la deuxième adresse symbolique est le numéro du registre dont le contenu des 24 positions les moins significatives représente l'« étiquette courante », b) la première adresse symbolique indique le numéro du registre dont on diminue le contenu d'une unité; si le résultat est nul, la valeur de l'« étiquette courante » est augmentée d'une unité, dans le cas contraire le contenu des 24 positions les moins significatives du registre indiqué par la deuxième adresse symbolique est pris comme « étiquette courante ».

La construction d'adresse du quatrième type s'utilise dans les opérateurs de branchement analogues aux opérateurs à construction du premier type à cette différence près qu'on prend pour l'« étiquette courante » non pas le contenu du registre indiqué par la deuxième adresse symbolique, mais la valeur même de cette adresse.

La construction d'adresse du cinquième type s'utilise dans les opérateurs qui réalisent certaines opérations arithmétiques spéciales et une opération de branchement en fonction du résultat de l'opération arithmétique. Le contenu du registre indiqué par la première adresse symbolique de la construction d'adresse s'additionne (d'après les règles d'addition en virgule fixe) avec le contenu du deuxième registre, le résultat s'inscrit à la place du premier opérande et représente une donnée initiale pour la deuxième opération qui effectue la comparaison de cette donnée avec le contenu d'un registre d'adresse impaire, égale à la valeur de la deuxième adresse symbolique de la construction (si elle est impaire) ou dépassant cette valeur d'une unité.

La comparaison se fait dans le but de vérifier l'une des conditions suivantes, selon la désignation symbolique de l'opérateur:

le résultat de l'addition est plus grand que le contenu du registre d'adresse impaire, ou bien ce résultat est au plus égal au contenu du registre. Si la condition est remplie, on prend pour l'« étiquette courante » la valeur de la troisième adresse symbolique de la construction d'adresse, dans le cas contraire la valeur de l'« étiquette courante » s'augmente d'une unité.

La construction d'adresse du deuxième type est utilisée dans l'opérateur de branchement inconditionnel qui impose pour valeur de l'« étiquette courante » le contenu des 24 positions les moins significatives du registre dont l'adresse est égale à la valeur de l'unique adresse symbolique de la construction d'adresse.

La construction d'adresse spéciale s'utilise dans les opérateurs de branchement conditionnel. Cette construction contient l'unique adresse symbolique qui désigne l'étiquette de l'opérateur auquel passe la commande. L'exécution de l'opérateur comprend la comparaison de l'indice de résultat avec une constante déterminée par la désignation symbolique de l'opérateur. S'il y a égalité, la valeur de l'adresse symbolique est prise pour celle de l'« étiquette courante », sinon cette dernière s'augmente d'une unité.

Dans le langage d'assemblage il existe un opérateur qui utilise une construction d'adresse du quatrième type et qui a des traits communs avec les opérateurs arithmétiques et ceux de contrôle, bien qu'il n'en soit pas un. Cet opérateur réalise l'extraction de la mémoire du contenu de la cellule indiquée par la deuxième adresse symbolique de la construction d'adresse et l'addition des positions 8 à 15 de cette cellule avec le contenu des positions 24 à 31 du registre indiqué par la première adresse symbolique de la construction d'adresse; le résultat n'est pas enregistré dans la mémoire, mais s'utilise en tant qu'instruction à exécuter immédiatement; après son exécution la valeur de l'« étiquette courante » s'augmente d'une unité.

6.6.2. Macro-opérateurs. Outre les opérateurs simples que nous venons de décrire la classe des opérateurs d'action du langage de programmation symbolique contient des macro-opérateurs.

Un macro-opérateur représente une suite de chaînes qui se composent chacune de trois éléments.

Le corps du premier élément de la première chaîne contient une adresse symbolique d'opérateur (une étiquette) ou un mot se composant de symboles || ⊔ ||. Le corps du deuxième élément de la première chaîne contient la désignation symbolique de l'opérateur qui diffère des désignations symboliques d'opérateurs simples. Les corps des premiers et deuxièmes éléments des chaînes suivantes se composant de symboles || ⊔ ||.

Le corps du troisième élément de chaque chaîne contient une construction d'adresse qui représente une suite finie d'adresses sym-

boliques d'opérandes séparées par les symboles || ( )||. Le nombre de chaînes qu'on utilise pour noter un macro-opérateur est le quotient, arrondi par excès à l'entier le plus proche, du nombre de symboles dans la suite des adresses symboliques par le nombre maximal de symboles qu'on peut écrire dans le corps du troisième élément de chaîne.

Un macro-opérateur détermine l'exécution d'une certaine suite d'opérateurs simples. Les données unitaires pour cette suite sont soit les opérandes isolés dont les adresses symboliques figurent dans la suite des adresses symboliques du macro-opérateur, soit des constructions d'opérandes isolés.

Le résultat d'exécution d'une suite d'opérateurs simples se présente également sous la forme d'un opérande isolé ou d'une construction d'opérandes isolés. La place de chaque opérande isolé dans la construction est déterminée par les adresses symboliques correspondantes indiquées dans le macro-opérateur.

Exemple 6.14. Dans le langage de codage symbolique du calculateur « Minsk-32 », il y a une construction spéciale « appel de programme » qui représente un macro-opérateur et a la forme d'une suite de chaînes dont chacune se compose de trois éléments. Le corps du premier élément de la première chaîne contient ou bien une adresse symbolique, ou bien les symboles || ⊔ ⊔ ⊔ ⊔ ⊔ ||. Le corps du deuxième élément de la première chaîne contient le mot || ИП ⊔ ⊔ ⊔ ||. Le corps du troisième élément de la première chaîne contient une suite qui commence par une adresse symbolique suivie du symbole || ; ||, puis on note ou bien une adresse symbolique, ou bien un nombre entier (décimal ou octal) écrit d'après les règles de notation des opérandes directs-littéraux (voir l'exemple 6.12).

Le nombre de chaînes suivantes est égal à la valeur du troisième terme de la suite écrite dans le corps du troisième élément de la première chaîne de l'appel de programme. Chacune des chaînes suivantes représente l'une des constructions possibles, dans le langage de codage symbolique de « Minsk-32 », de définition d'état de cellules de mémoire (voir l'exemple 6.8).

Le macro-opérateur d'appel de programme provoque l'exécution d'une suite déterminée d'opérateurs simples.

Cette suite est donnée (voir l'exemple 6.16) par le premier terme de la suite écrite dans le corps du troisième élément de la première chaîne de l'appel de programme. Les opérandes « extérieurs » pour la suite d'opérateurs simples sont donnés dans la deuxième chaîne et les chaînes suivantes de l'appel de programme.

Dans le langage de codage symbolique il existe également des opérateurs d'échange d'information. Chacun d'eux appelle le moniteur qui exécute une suite d'opérateurs réalisant l'échange d'information entre la mémoire rapide et la mémoire extérieure.

Un opérateur d'échange contient deux ou trois chaînes. La première se compose de trois éléments. Le corps du premier élément peut contenir une adresse symbolique d'opérateur ou bien des symboles || ⊔ ||, dans le corps du deuxième on écrit la désignation symbolique de l'opérateur, le corps du troisième élément contient dans l'ordre : un symbole || : ||, l'adresse symbolique d'une cellule d'index, un symbole || ; ||, l'indice d'opération (exprimé soit par un mot de une à trois lettres, soit par un nombre binaire à sept chiffres mis entre parenthèses), un symbole || ; || et la désignation symbolique d'un dispositif extérieur.

Dans la deuxième chaîne de l'opérateur d'échange on écrit une « constante d'échange » qui indique quand il faut terminer l'échange (en rencontrant un symbole ou un mot donné), et l'on donne l'adresse symbolique de la cellule initiale de la zone d'échange (et le numéro du symbole initial de cette cellule). La constante d'échange sert à former un mot de commande que le moniteur retient.

Si un opérateur d'échange s'exécute après un autre opérateur d'échange et que leurs mots de commande soient les mêmes, alors on écrit dans la deuxième chaîne du dernier opérateur un indicateur, d'après lequel l'ancien mot de commande est réutilisé.

Le programmeur se sert de la troisième chaîne lorsqu'il prévoit des méthodes spéciales de traitement des indicateurs d'état de dispositifs extérieurs. On y écrit les désignations symboliques des indicateurs interrogés.

Exemple 6.15. Dans le langage d'assemblage du système IBM-360, il y a des constructions, appelées macro-instructions, qui déterminent l'exécution d'une suite d'opérateurs d'action simples.

Une macro-instruction représente une suite de chaînes de sept éléments chacune.

Le corps du premier élément de la première chaîne contient ou bien une adresse symbolique, ou bien huit symboles || ⊔ ||. Le corps du deuxième élément de chaque chaîne contient le symbole || ⊔ ||. Le corps du troisième élément dans la première chaîne contient la désignation symbolique de la macro-instruction, et dans toutes les chaînes suivantes, le mot || ⊔ ⊔ ⊔ ⊔ ⊔ ||. Le corps du quatrième élément de chaque chaîne contient le symbole || ⊔ ||. Le corps du cinquième élément de chaque chaîne contient une construction appelée indicateur d'opérandes qui se compose d'indicateurs d'opérandes élémentaires séparés par des symboles || , ||. Le nombre d'indicateurs d'opérandes élémentaires ne doit pas dépasser 100. Chaque indicateur élémentaire peut contenir de 0 à 127 symboles. Le nombre de chaînes qu'on utilise pour écrire une macro-instruction est le quotient arrondi par excès à l'entier le plus proche, du nombre de symboles dans la suite des indicateurs d'opérandes par 56 (le nombre

de symboles qu'on peut écrire dans le corps du cinquième élément de chaîne).

Chaque indicateur d'opérandes élémentaire représente une construction de symboles arbitraires de l'alphabet du langage d'assemblage soumise à certaines restrictions. Les symboles || ′ || (guillemet) et || (||, ||) || doivent aller par paires (remarquons qu'un symbole || ″ || est considéré comme un symbole || ′ ||, et qu'à chaque symbole || (|| doit correspondre un symbole ||) || disposé plus à droite). Le symbole || = || peut être écrit ou bien comme premier symbole d'un opérande, ou bien entre deux symboles || ′ || ou || (|| et ||) || formant un couple. Un indicateur d'opérandes où se suivent deux symboles || , || signifie que l'indicateur élémentaire est « vide », i.e. ne contient aucun symbole.

Les restrictions évoquées n'empêchent pas d'utiliser en tant qu'indicateurs élémentaires d'opérandes les adresses symboliques, les opérandes directs (donnés en formes décrites dans l'exemple 6.9), ainsi que les listes d'indicateurs qui sont des constructions qui commencent par un symbole || (||, se terminent par un symbole ||) || et contiennent entre ces symboles une suite d'indicateurs d'opérandes élémentaires séparés par des symboles || , ||.

Le corps du sixième élément de chaque chaîne, sauf la dernière, contient l'un des symboles de l'alphabet, à l'exception de || ⊔ ||. Le symbole || ⊔ || est contenu dans le corps du sixième élément de la dernière chaîne. Le corps du septième élément de chaque chaîne contient les symboles || ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ ||.

6.6.3. Opérateurs de description des constructions d'opérateurs. Les constructions d'opérateurs sont décrites au moyen de deux opérateurs spéciaux, un opérateur initial et un opérateur final.

L'opérateur initial de description des constructions d'opérateurs se compose de trois éléments de chaîne. Le corps du premier élément contient une suite de symboles || ⊔ ||, celui du deuxième contient le mot || PROGRAMME ||, celui du troisième contient une suite arbitraire de symboles de l'alphabet du langage (qui n'est pas une suite des seuls symboles || ⊔ ||).

L'opérateur final de description des constructions d'opérateurs se compose lui aussi de trois éléments de chaîne. Le corps du premier d'entre eux contient une suite de symboles || ⊔ ||, le corps du deuxième contient le mot || FIN ||, dans celui du troisième est reproduite la même suite de symboles qui figure dans le corps du troisième élément de l'opérateur initial.

A la suite d'un opérateur initial on écrit une suite finie d'opérateurs, puis l'opérateur final correspondant. Les constructions d'opérateurs mises entre un opérateur initial et un opérateur final sont de deux types: constructions correspondant à des macro-opérateurs et constructions-blocs.

17—0779

Il s'agit d'une construction correspondant à un macro-opérateur, lorsqu'une autre suite d'opérateurs contient un macro-opérateur dont le corps du deuxième élément de la première chaîne représente la même suite de symboles qui est écrite dans le corps des troisièmes éléments des opérateurs initial et final décrivant cette construction. La construction correspondant à un macro-opérateur doit contenir les opérateurs dont les adresses symboliques d'opérandes sont de la forme : le symbole ‖ A‖, un symbole ‖ '‖, une suite de chiffres exprimant un entier décimal sans signe, un symbole ‖ '‖. L'entier décimal peut varier de 1 à n, où n est le nombre d'indicateurs élémentaires d'opérandes dans le macro-opérateur auquel correspond la construction d'opérateurs.

Une construction-bloc est une construction d'opérateurs telle que la suite de symboles écrite dans les corps des troisièmes éléments des opérateurs initial et final de sa description ne se trouve dans aucun macro-opérateur, dans le corps de son deuxième élément de la première chaîne.

Une construction correspondant à un macro-opérateur se réalise de la façon suivante : au cours de l'exécution du programme en langage machine, une fois exécutées les instructions correspondant aux opérateurs d'action qui précèdent le macro-opérateur, on exécute la suite d'instructions correspondant aux opérateurs de la construction ; ensuite vient le tour des instructions correspondant aux opérateurs qui suivent le macro-opérateur. Lors de la traduction des opérateurs d'une construction correspondant au macro-opérateur, les adresses symboliques d'opérandes qui contiennent un entier sans signe k entre les symboles ‖ '‖ et ‖ '‖ sont remplacées par la valeur du k-ième indicateur élémentaire d'opérande du macro-opérateur.

Les constructions-blocs sont des constructions indépendantes en ce sens que les programmes qui leur correspondent peuvent être rangés de façon quelconque dans la mémoire. La correspondance entre les opérandes et les opérateurs de tels blocs est établie grâce à l'utilisation, dans les différents blocs, des mêmes adresses symboliques d'opérandes et d'opérateurs (étiquettes).

Exemple 6.16. Dans le langage de codage symbolique du calculateur « Minsk-32 », sont définies des constructions d'opérateurs qui correspondent aux macro-opérateurs — appels des programmes (voir l'exemple 6.14).

Une telle construction représente une suite d'opérateurs d'action précédée d'un opérateur initial de description qui contient $l+1$ chaînes ($l \geqslant 1$). La première chaîne comprend le mot ‖ ВХОД‖ dans le corps du premier élément, le mot ‖ РЗВ ⊔ ⊔‖ dans le corps du deuxième, et le chiffre ‖ 3‖ dans le corps du troisième élément. Les chaînes suivantes contiennent des adresses symboliques dans le corps du premier élément, le mot ‖ РЗВ ⊔ ⊔‖ dans le

corps du deuxième et des nombres décimaux entiers dans le corps du troisième élément.

La somme de tous les nombres indiqués dans les troisièmes éléments de la deuxième chaîne et des chaînes suivantes doit être égale à la valeur du troisième terme de la suite qui est écrite dans le corps du troisième élément de la première chaîne du macro-opérateur d'appel de programme (auquel correspond la construction décrite par l'opérateur initial).

Les opérateurs d'action de la construction qui utilisent des opérandes extérieurs, contiennent, en tant qu'adresses symboliques de ces opérandes, les adresses symboliques indiquées dans les corps des premiers éléments de la deuxième chaîne et des chaînes suivantes de l'opérateur initial de description de construction. Si, dans la m-ème chaîne ($1 < m \leqslant l + 1$), le nombre indiqué dans le troisième élément est supérieur à l'unité, alors, pour les opérandes successifs à partir de celui qui est donné dans la r-ème chaîne de l'appel de programme (où r est la somme des nombres dans les troisièmes éléments des chaînes à partir de la 2-ème jusqu'à la m — 1-ème de l'opérateur initial de description), on utilise les adresses symboliques α; $\alpha + 1$; $\alpha + 2$; . . ., $\alpha + k - 1$ où α est l'adresse symbolique donnée dans le premier élément de la m-ème chaîne.

L'opérateur final de description d'une construction suit ses opérateurs d'action et se compose d'une seule chaîne. Le corps de son premier élément contient une suite de symboles || ⊔ ||, le corps du deuxième, le mot || ВЫХ ⊔ ⊔ ||, le corps du troisième, les symboles || ВХОД ; || suivis soit d'une adresse symbolique, soit d'un nombre, identique à l'adresse symbolique ou au nombre indiqué à la troisième place de la suite écrite dans le corps du troisième élément de la première chaîne du macro-opérateur appel de programme auquel correspond la construction décrite par les opérateurs de description initial et final.

Si la construction d'opérateurs qui correspond à un appel de programme représente un « programme de bibliothèque », alors la suite d'opérateurs qui contient de tels appels doit être complétée par une construction « description de programme » qui occupe deux chaînes. Dans le corps du premier élément de la première chaîne on écrit une adresse symbolique, dans le corps du deuxième, le mot || ОПР ⊔ ⊔ ||, dans le corps du troisième, une suite de symboles qui est le nom du « programme de bibliothèque ». Dans la deuxième chaîne, le corps du deuxième élément contient le mot || НОП ⊔ ⊔ ||, ceux du troisième et du premier contiennent seuls les symboles || ⊔ ||.

Il existe, dans le langage, les constructions « description de programme spéciale » et « description de programme de chargement ». La première s'emploie lorsque dans la construction à laquelle correspond la « description de programme spéciale », on utilise comme opérande ou bien la même construction « description de

programme spéciale », ou bien son adresse initiale. Cette construction diffère de celle que nous venons de décrire par ce qu'à la place de || ОПР ␣ ␣ ||, on met || ОПРС ␣ ||. Une « description de programme de chargement » diffère d'une « description de programme » par le fait de comporter trois chaînes dont la deuxième et la troisième sont identiques.

Si pour une construction d'opérateurs décrite par un opérateur de description initial et un opérateur final, on a encore l'une des « descriptions de programme » indiquées, alors, dans les opérateurs d'appel qui correspondent à ce programme, la valeur du troisième terme de la suite dans le troisième élément de la première chaîne doit être égale à l'adresse de la cellule où est gardée la première chaîne de la « description de programme ». Dans le cas contraire la valeur de l'élément indiqué doit être égale à l'adresse, diminuée d'une unité, de la cellule gardant la première chaîne de l'opérateur initial de description de la construction d'opérateurs qui correspond à l'appel de programme.

Exemple 6.17. Dans le langage d'assemblage du système IBM-360, il existe deux classes d'opérateurs de description des constructions d'opérateurs: descripteurs de modules de programme et descripteurs de macrodéfinitions.

Un module est en principe analogue à une construction-bloc, et une macrodéfinition est analogue à une construction correspondant à un macro-opérateur (§ 6.8). Un module représente une suite d'opérateurs d'action précédée d'un opérateur initial de description de ce module et suivie d'un opérateur final. Chaque module peut comporter plusieurs sections de commande. On définit une section comme une suite d'opérateurs d'action du module précédée de l'opérateur initial de description de section. L'opérateur initial d'un module est en même temps l'opérateur initial de sa première section de commande.

Un opérateur initial de description du module de programme est une chaîne de sept éléments. Le corps du premier élément contient ou bien une adresse symbolique élémentaire, ou bien une suite de symboles || ␣ ||. Dans le corps du troisième élément on place le mot || START ||, le corps du cinquième contient ou bien une adresse symbolique exprimée par un argument d'adresse, ou bien une suite de symboles || ␣ ||. Les corps des deuxième, quatrième, sixième et septième éléments contiennent uniquement des symboles || ␣ ||.

Un opérateur final de description du module de programme représente une chaîne dont le corps du troisième élément contient le mot || END ||, les corps des autres éléments ne contenant que des symboles || ␣ ||.

Un opérateur initial de section de commande représente une chaîne dont le corps du premier élément contient ou bien une adresse

symbolique élémentaire, ou bien une suite de symboles || ⊔ ||. Le corps du troisième élément contient le mot || CSECT ||, les corps des autres éléments ne contenant que des symboles || ⊔ ||.

On appelle section de commande réunie l'ensemble des corps de toutes les sections de commande qui contiennent les mêmes symboles dans les corps des premiers éléments de leurs opérateurs initiaux.

Dans le cas particulier où un module de programme comporte une seule section de commande, ce bloc peut ne pas avoir d'opérateur de description initial.

Les opérateurs initiaux de sections de commande établissent les noms de sections et permettent à l'assembleur de réunir plusieurs sections en des suites continues d'opérateurs. L'opérateur initial de description d'un module de programme, outre qu'il donne le nom de la première section de commande, peut indiquer par l'adresse symbolique dans le corps du cinquième élément de sa chaîne l'étiquette de celui des opérateurs d'action du module qui s'exécute le premier. Dans le cas où le corps du cinquième élément ne contient que des symboles || ⊔ ||, le premier à exécuter est le premier opérateur de la suite. La situation est la même dans les cas où un opérateur initial de section est écrit au lieu de l'opérateur initial du module, ou bien lorsque celui-ci n'est pas mentionné du tout. Les deux formes de notation sont autorisées.

Les opérateurs-descripteurs de macrodéfinitions font partie d'un sous-langage du langage d'assemblage, i.e. du langage des macrodéfinitions. Dans le présent exemple nous n'en donnons que des fragments.

Une macrodéfinition est une construction d'opérateurs d'action du langage d'assemblage qui correspond à une macro-instruction. Dans le software IBM-360 il y a certaines macro-instructions dont on peut se servir en programmant dans le langage d'assemblage. Ces macro-instructions correspondent au langage du système exécutif formé par les opérateurs réalisés par les circuits de la machine et par programme. Dans le software, à de telles macro-instructions il correspond des macrodéfinitions.

Outre la possibilité de se servir des macro-instructions correspondant aux macrodéfinitions déjà existantes, l'utilisateur du langage d'assemblage peut composer de nouvelles macro-instructions et donc les macrodéfinitions respectives. Dans ce travail il faut employer le langage des macrodéfinitions.

Dans une macrodéfinition on distingue les parties suivantes: 1) un opérateur de début de macrodéfinition, 2) un opérateur du prototype de macro-instruction, 3) un modèle de macrodéfinition, 4) un opérateur de fin de macrodéfinition.

L'opérateur de début de macrodéfinition se compose de sept éléments, le corps du troisième contient le mot || MACRO ||, les corps des autres ne contenant que des symboles || ⊔ ||.

Le prototype de macro-instruction représente une construction qui coïncide avec celle de la macro-instruction (voir l'exemple 6.15) à laquelle correspond la macrodéfinition, à ceci près que: le corps du premier élément de la première chaîne du prototype peut contenir ou bien une suite de symboles || ␣ ||, ou bien l'adresse symbolique élémentaire, dont le premier symbole est ||&||, qui s'appelle « adresse symbolique variable ». Les corps des cinquièmes éléments des chaînes contiennent un indicateur d'opérandes dans lequel chaque indicateur élémentaire commence par le symbole ||&||. Le corps du troisième élément contient le même code symbolique de macro-instruction qui est indiqué dans la macro-instruction correspondant à la macrodéfinition donnée.

Le modèle de macrodéfinition représente une suite d'opérateurs d'action du langage d'assemblage, parmi lesquels il y a des opérateurs contenant en tant que les adresses symboliques (étiquettes), ou les désignations symboliques, ou les adresses symboliques d'opérandes dans les indicateurs d'opérandes les mots tels que les adresses symboliques variables employées dans le prototype de macro-instruction (qui commencent par le symbole ||&||) en sont des occurrences.

L'opérateur de fin de macrodéfinition diffère de celui de début par le fait de comporter dans le corps du troisième élément le mot || MEND || à la place de || MACRO ||.

Lorsqu'une macro-instruction est rencontrée dans le programme d'une section de commande, l'assembleur effectue l'identification de l'instruction avec la macrodéfinition, puis assure l'exécution de la macrodéfinition, ayant remplacé préalablement, dans le modèle, les occurrences qui commencent par le symbole ||&|| par certaines constructions dépendant des indicateurs élémentaires d'opérandes donnés dans la macro-instruction et de l'étiquette de celle-ci.

Il existe les différentes manières de réaliser un tel remplacement. La solution la plus simple consiste à substituer à une adresse symbolique variable α qui commence par un ||&||, un indicateur élémentaire d'opérande (ou d'une étiquette) pris dans la macro-instruction. La place de cet indicateur élémentaire dans la macro-instruction correspond à la place de la construction α dans son prototype. On dit alors qu'une « compilation simple de macrodéfinition » a lieu. Si l'un des indicateurs élémentaires d'opérandes dans le prototype de la macro-instruction a la forme ||& NUM||, alors la même place dans l'indicateur d'opérandes de la macro-instruction est occupée par une liste et dans les opérateurs d'action du modèle on doit rencontrer des constructions de la forme || & NUM (|| l ||) || où l est un entier qui indique l'élément de liste utilisé dans l'opérateur donné.

Dans les cas plus compliqués de « compilation conventionnelle », le modèle de macrodéfinition contient, entre le prototype de macro-

instruction et les opérateurs d'action, les « instructions de compilation conventionnelle » qui donnent certaines fonctions, dont les arguments sont les adresses symboliques dans la macro-instruction et dont les valeurs sont mises à la place des adresses symboliques variables dans les opérateurs d'action du modèle. Parmi les opérateurs d'action du modèle, il peut se trouver un opérateur qui contient dans le corps du troisième élément le mot ‖ MEXIT‖, désignant la cessation de l'exécution des opérateurs de la macrodéfinition lors de sa compilation.

6.6.4. Opérateurs de description et d'allocation de mémoire. Les opérateurs de description et d'allocation de mémoire sont représentés dans le langage de programmation symbolique par les opérateurs qui déterminent les états des zones de mémoire réservées au opérandes ainsi que par les opérateurs spéciaux destinés à implanter en mémoire des suites d'opérateurs d'action.

Les opérateurs qui déterminent l'état des emplacements de mémoire sont décrits au § 6.5 comme appartenant au langage des opérandes.

Un opérateur de réservation de mémoire pour une suite d'opérateurs (un bloc) se compose de trois éléments de chaîne. Le corps du premier comporte uniquement des symboles ‖⊔ ‖. Le corps du deuxième contient le mot ‖ DEBUT‖, celui du troisième, une construction se composant d'une suite de symboles mise dans le corps du troisième élément des opérateurs initial et final de la description du bloc, d'un symbole ‖;‖ et d'une adresse symbolique.

L'opérateur de réservation de mémoire dispose les opérateurs du bloc à implanter à partir de l'emplacement élémentaire de mémoire dont l'adresse est égale à la valeur de l'adresse symbolique indiquée à la troisième place de la construction écrite dans le troisième élément de chaîne de cet opérateur. On détermine ainsi les valeurs des adresses de tous les opérateurs du bloc (de toutes les étiquettes) et des adresses des opérandes-objets qui se rencontrent dans la suite d'opérateurs.

Exemple 6.18. Il existe dans le langage de codage symbolique du calculateur « Minsk-32 » des opérateurs de description et d'allocation de mémoire. Ce sont les opérateurs qui attribuent à des emplacements élémentaires de mémoire les états représentant les opérandes (voir les exemples 6.8 et 6.10), ainsi que les opérateurs de rangement dans la mémoire des constructions d'opérandes.

Pour mettre dans la mémoire une construction d'opérateurs qui correspond au macro-opérateur « appel de programme », on écrit devant l'opérateur initial de desription d'une telle construction un opérateur qui établit l'adresse relative du premier opérateur de la construction. Cet opérateur se compose de trois éléments de chaîne

dont le premier contient seuls les symboles || ⊔ ||, le deuxième, le mot ||БАЗ||, le troisième le symbole ||0||.

Les programmes rédigés dans le langage de codage symbolique mais ne correspondant pas à un appel de programme (à un macro-opérateur), peuvent être disposés dans les zones de mémoire de trois types: une zone principale, des zones de manœuvre, des zones communes. Chaque programme utilise une zone principale et un nombre arbitraire de zones de manœuvre et de zones communes. La capacité de la zone principale ne dépasse pas 2048 cellules, celle des autres zones va jusqu'à 65 536 cellules.

On peut ranger dans une zone ou bien une suite d'opérateurs d'action, ou bien une suite d'opérateurs de rangement des opérandes dans la mémoire (voir l'exemple 6.8).

Remarquons que l'opérateur de description d'un programme (ayant la désignation symbolique ||ОПР||, exemple 6.16) ne peut se trouver que dans la zone principale.

Pour chaque zone, on utilise un opérateur qui donne l'adresse initiale de la zone. Cet opérateur se compose de trois éléments de chaîne. Dans le corps du premier on écrit ou bien une adresse symbolique, ou bien une suite de symboles || ⊔ ||; le dernier cas est obligatoire pour une zone principale. Dans le corps du deuxième élément on écrit le mot ||БАЗ||, dans le corps du troisième, le symbole ||0|| pour une zone principale et un entier sans signe ou une adresse symbolique suivi du symbole ||;||, suivi du nom de zone exprimé par les mots ||РАБ|| ou ||ОБЩ|| suivant qu'il s'agit d'une zone de manœuvre ou d'une zone commune.

Dans l'opérateur indiqué, le nombre ou l'adresse symbolique qu'on écrit à la première place dans le troisième élément de chaîne indique la cellule de « base » dont le contenu représente l'adresse de la première cellule de la zone.

Remarquons que l'adresse de la cellule initiale d'une zone principale est toujours nulle, et que chaque construction d'opérateurs qui correspond à un appel de programme est censée d'être rangée dans une zone principale.

Pour les zones allouées, il est possible d'indiquer, à l'aide d'un opérateur spécial, l'adresse de l'opérateur d'action à exécuter le premier dans une zone donnée. Cet opérateur spécial ne contient que des symboles || ⊔ || dans le corps du premier élément, le mot ||НАЧ|| dans le corps du deuxième et une adresse symbolique dans le corps du troisième élément. La valeur de cette adresse symbolique est l'adresse de ce premier opérateur par rapport au début de la zone.

On utilise également un opérateur qui permet de désigner certaines parties de zones par des adresses symboliques spéciales. Cet opérateur se compose de trois éléments de chaîne et se situe dans la zone dont il faut désigner une partie. Le corps de son premier élément contient une adresse symbolique qui sert de nom à la partie

en question, le corps du deuxième, le mot || P3B || et le corps du troisième, un entier sans signe octal (avec le symbole || B || après ses chiffres) ou décimal, qui indique le nombre de cellules dans la séquence séparée.

Il y a aussi un opérateur qui donne le nombre de cellules utiliseés par un programme en tant que registres d'index. Cet opérateur peut être donné dans n'importe quelle partie du programme et se compose de trois éléments de chaîne. Le premier ne contient que des symboles || ␣ ||, le deuxième contient le mot || РИП ||, le troisième, un entier sans signe, multiple de seize, octal (avec le symbole || B ||) ou décimal. Un programme peut contenir n'importe quel nombre de tels opérateurs. En rencontrant un tel opérateur dans le programme, le dispatcher sait qu'il faut effectuer une réservation (ou modifier des réservations déjà faites) de mémoire pour les cellules-registres d'index.

EXEMPLE 6.19. Dans le langage d'assemblage IBM-360, une partie des opérateurs de description et d'allocation de mémoire se rapporte au langage des opérandes (opérateurs avec la désignation symbolique || DC || dans les exemples 6.9 et 6.11). Ces opérateurs réalisent l'enregistrement des opérandes dans les emplacements élémentaires de mémoire. Il existe d'autres opérateurs, de nature semblable, qui, sans enregistrer l'information dans la mémoire « réservent » les cellules pour que les opérateurs d'action correspondants l'y enregistrent par la suite. Ces opérateurs diffèrent des opérateurs || DC || par leur désignation qui est || DS ||. En les exécutant l'assembleur ne modifie pas l'état des cellules résersées.

Les opérateurs initiaux et finaux de description des constructions d'opérateurs (exemple 6.17) sont encore des opérateurs de description et d'allocation de mémoire. Outre la description de sections de commande qui se composent d'opérateurs, le langage d'assemblage prévoit une possibilité de donner des sections « vides » dont le contenu est déterminé au cours de l'exécution du programme. On utilise pour cela un opérateur qui diffère de l'opérateur déjà décrit avec la désignation symbolique || CSECT || par sa désignation symbolique qui est || DSECT ||. Une section « fictive » peut être partagée en des parties plus petites au moyen des opérateurs || DS ||.

S'il y a plusieurs modules de programme, le langage prévoit la possibilité de donner une section de commande (non nommée) commune à tous les modules au moyen d'un opérateur qui contient dans le corps du troisième élément de chaîne le mot || COM || et dans les autres éléments, seuls les symboles || ␣ ||. L'étendue de la zone de mémoire à réserver à la section de commande commune est égale à la longueur de la plus grande de toutes les sections communes dans tous les modules. La section de commande commune est divisée en parties au moyen d'opérateurs avec les désignations symboliques

|| DC|| ou || DS||. Les opérateurs de tous les modules s'adressent aux opérandes de la section de commande commune en utilisant les adresses symboliques des opérateurs || DC|| ou || DS||.

Pour les sections de commande d'un module de programme, on établit les adresses relatives d'opérateurs à l'aide des opérateurs spéciaux ayant les désignations symboliques || USING|| et || DROP||.

Une instruction || USING||, représentant une suite de chaînes de sept éléments chacune, contient dans le corps du troisième élément de la première chaîne le mot || USING||, et dans les corps des cinquièmes éléments, une suite contenant de deux à dix-sept adresses symboliques séparées par des virgules || ,||. La première de ces adresses peut être absolue ou translatable, les autres étant absolues.

Les adresses symboliques, à partir de la deuxième, désignent les registres que l'on peut utiliser en tant que registres de base dans une section de commande. La valeur A de la première adresse symbolique est la quantité qui doit être mise dans le registre ayant l'adresse symbolique indiquée à la deuxième place dans la suite d'adresses. Chacun des registres suivants doit contenir la quantité $A + (l - 1)\,4096$, où A est la valeur de la première adresse symbolique, l le numéro de l'adresse symbolique dans la suite ($2 \leqslant l \leqslant 17$). Remarquons que l'opérateur décrit ne range pas les valeurs indiquées dans les registres, mais détermine seulement ceux des registres qui seront utilisés en tant que registres de base au cours de l'exécution des opérateurs d'action qui suivent l'opérateur donné. L'enregistrement des valeurs dans les registres doit être organisé par le programmeur au moyen d'opérateurs d'action spéciaux (qui précèdent en général l'opérateur || USING||).

Tous les autres éléments des chaînes de l'opérateur décrit contiennent seuls les symboles || ⊔ ||, à l'exception du sixième élément, qui contient || ⊔ || seulement dans la dernière chaîne, toutes les autres chaînes contenant à cet endroit n importe quel symbole, sauf || ⊔ ||.

Il existe un opérateur de « démission » des registres de base. Il diffère de l'opérateur || USING|| par la présence dans le corps du troisième élément du mot || DROP|| à la place de || USING|| et par ce que le corps du cinquième élément contient de 1 à 16 adresses symboliques absolues. Cet opérateur montre que, dans les opérateurs d'action qui le suivent, les registres dont il donne les adresses symboliques ne seront pas utilisés comme registres de base.

Le langage d'assemblage comprend aussi les opérateurs décrivant les branchements vers les différents opérateurs d'action des sections de commande qui font partie de modules différents, ces modules étant séparés l'un de l'autre dans la mémoire. Les étiquettes de tous les opérateurs d'une section de commande auxquels peut passer le contrôle à partir des sections de commande d'autres modules, sont mentionnées dans un opérateur qui représente une suite de

chaînes de sept éléments chacune. Le corps du troisième élément de la première chaîne contient le mot || ENTRY ||, les corps des cinquièmes éléments contiennent une suite de 1 à 100 adresses symboliques translatables d'opérateurs (d'étiquettes) de la section de commande à laquelle appartient l'opérateur donné, ces adresses étant séparées par des symboles || , ||. Les corps des autres éléments des chaînes contiennent seuls les symboles || ␣ ||, à l'exception du sixième élément qui contient || ␣ || dans la dernière chaîne seulement, et n'importe quel symbole sauf || ␣ || dans toutes les autres chaînes.

Dans une section de commande qui contient un opérateur || ENTRY ||, on écrit aussi un opérateur ayant la désignation symbolique || EXTRN || dans lequel les cinquièmes éléments des chaînes contiennent une suite de 1 à 100 étiquettes d'opérateurs (d'adresses symboliques translatables) d'autres sections de commande « extérieures » auxquelles la section donnée peut passer le contrôle du programme. Les étiquettes indiquées dans l'opérateur || EXTRN || ne doivent coïncider avec aucune étiquette utilisée dans la section de commande qui contient cet opérateur.

Les opérateurs || EXTRN || et || ENTRY || contiennent des fragments du dictionnaire des passages de commande entre sections utilisé par l'éditeur des liens qui fait partie du software IBM-360. Remarquons que, s'il y a branchement d'une section de commande vers le début d'une autre section de commande qu'on recherche par son nom, alors ce nom n'est pas écrit dans l'opérateur || ENTRY ||, car les noms de toutes les sections de commande sont automatiquement mis dans le dictionnaire des passages.

Remarquons encore que, pour organiser les passages de commande entre sections, l'utilisation des opérateurs || EXTRN || et || ENTRY || n'est pas obligatoire. On peut se servir de l'opérande-adresse qui commence par le symbole || V || (voir l'exemple 6.9) et dont la valeur est égale à l'étiquette de l'opérateur de la section de commande « extérieure » auquel doit passer la commande. Cet opérande-adresse est mis par un opérateur d'action dans un registre, et un autre opérateur d'action réalise le branchement vers l'adresse contenue dans le registre.

6.6.5. Opérateurs de mise au point. Nous allons considérer trois opérateurs de mise au point. Chacun se compose de trois éléments de chaîne. Le premier opérateur contient dans le corps de son premier élément uniquement des symboles || ␣ ||, le corps du deuxième élément contient le mot || IMPRESSION ||, le corps du troisième, une adresse symbolique d'opérateur suivie du symbole || , || suivi d'une adresse symbolique d'opérateur. Le deuxième opérateur diffère du précédent par son deuxième élément contenant le mot || PERFORATION || au lieu de || IMPRESSION ||.

Le troisième opérateur contient dans le corps de son deuxième élément de chaîne le mot || EXECUTER ||, les autres éléments étant organisés comme dans les opérateurs précédents.

Les deux premiers opérateurs permettent d'imprimer ou de perforer les instructions qui correspondent à la suite d'opérateurs commençant par l'opérateur dont l'adresse occupe la première place dans le troisième élément de chaîne et finissant par l'opérateur dont l'adresse est écrite à la deuxième place dans le même élément.

Le troisième opérateur assure l'exécution, après assemblage, de la même suite d'opérateurs.

Les opérateurs décrits fournissent l'information sur le programme en instructions de machine, le programme même perforé sur un support mécanographique ou les résultats de son exécution.

Exemple 6.20. Dans le langage de codage symbolique il existe un groupe d'opérateurs qui permettent de sortir de la machine, un programme compilé sous une forme déterminée par les règles d'édition, c'est-à-dire, avec les en-têtes, les commentaires, les sauts de lignes demandés, etc.

En plus, il existe des opérateurs permettant de communiquer à l'assembleur une information supplémentaire, ou de recevoir une telle information du programme.

L'information supplémentaire sur l'utilisation d'adresses symboliques dans différentes parties du programme est transmise à l'assembleur au moyen de l'opérateur « liste d'étiquettes disponibles ». Lorsque cet opérateur est disposé à la fin d'un bloc de programme, on y indique seules les adresses symboliques utilisées à l'intérieur du bloc et ne reliant pas entre eux les opérateurs et opérandes appartenant à des blocs différents. On peut utiliser ces mêmes adresses symboliques comme adresses locales dans d'autres blocs.

L'opérateur « transmettre l'adresse de l'indicateur d'occupation de mémoire » fournit une information sur l'exécution d'un programme compilé. Il permet au programmeur d'évaluer le volume de mémoire disponible.

Exemple 6.21. Dans le langage d'assemblage IBM-360 il existe des opérateurs permettant de perforer et d'imprimer un programme dans le langage machine. En tant que particularités de ces opérateurs, citons les possibilités de perforer le nom de module de programme sur chaque carte, de changer de page au cours de l'impression, de sauter les lignes, de fournir un module de programme soit complètement, soit sans opérateurs engendrés par des macro-instructions, soit sans constantes, etc. L'entrée d'un programme dans le langage d'assemblage, sa traduction dans le langage machine et son exécution s'effectuent à l'aide des cartes spéciales de commande qui se rapportent aux moyens de conversation de l'opérateur humain avec

le système IBM-360. De plus, on utilise des opérateurs supplémentaires qui permettent d'interpréter les perforations dans les cartes d'entrée (qui contiennent un programme dans le langage d'assemblage) d'une façon non standard. Il y a aussi un opérateur vérifiant si la succession des cartes d'entrée est correcte. Remarquons que l'on utilise pour cet opérateur le contenu des corps des septièmes éléments de chaîne des opérateurs contrôlés du langage d'assemblage. Dans les exemples précédents on supposait qu'ils contiennent seuls les symboles ‖ ⊔ ‖. Lorsqu'on pense vérifier la succession des cartes d'entrée, on écrit dans les corps de ces éléments les numéros des opérateurs et chaque erreur dans l'ordre de succession sera signalée.

§ 6.7. Exécution des notations dans le langage de programmation symbolique

En décrivant le langage algorithmique de programmation symbolique il faut considérer les règles d'interprétation des notations dans ce langage, i.e. les règles qui permettent d'exécuter successivement les opérateurs du langage et d'utiliser ses opérandes. On comprend l'exécution des notations en ce sens qu'on suppose l'existence d'un automate qui perçoit les notations telles quelles, sans les traduire ni transformer, et organise leur exécution.

L'exécution des notations dans le langage de programmation symbolique a beaucoup de traits communs avec l'exécution des programmes dans le langage machine. De même que dans le langage machine, après l'exécution d'un opérateur arithmétique isolé, il se produit le passage à l'opérateur d'action suivant. Chaque opérateur en cours d'exécution a son étiquette explicite ou implicite enregistrée dans la mémoire « étiquette courante ». Au début de l'exécution d'une notation dans le langage de programmation symbolique, on place dans la mémoire « étiquette courante » l'adresse symbolique du premier opérateur d'action. (L'« étiquette courante » dite aussi « compteur d'assemblage », est analogue du registre-compteur ordinal utilisé au moment de l'exécution du programme dans le langage machine.) Lorsqu'un opérateur à exécuter possède une étiquette, celle-ci est transférée dans le « compteur d'assemblage ». Lors de l'exécution d'un opérateur sans étiquette, on place dans le « compteur d'assemblage » la construction $N + (n + 1)$ (« étiquette implicite »), où N est l'adresse symbolique représentant la dernière étiquette explicite mise dans le « compteur d'assemblage », $(n + 1)$ est un entier sans signe dans lequel n désigne le nombre d'opérateurs exécutés après l'opérateur d'étiquette N.

En exécutant un passage de commande, on met dans la mémoire « étiquette courante » ou bien l'étiquette explicite de l'opérateur auquel passe la commande, ou bien l'étiquette (explicite ou implicite) de l'opérateur suivant.

Au cours de l'exécution de chaque opérateur, les opérations qui lui correspondent sont exécutées sur les opérandes indiqués par l'opérateur. Il faut que chaque opérande — donnée initiale soit « défini » avant son utilisation dans les opérateurs précédents. Nous considérons comme défini un opérande dont l'adresse symbolique ou bien désigne un opérande — résultat d'un quelconque des opérateurs précédents, ou bien s'utilise dans un opérateur précédent ayant la désignation symbolique || AFFECTER || dans le corps de son premier élément de chaîne.

Les opérateurs || AFFECTER || s'exécutent comme les opérateurs d'action en mettant les cellules correspondantes à l'état nécessaire, mais leur exécution n'entraîne pas la modification du « compteur d'assemblage ».

Au cours de l'exécution d'une macro-opération, la construction correspondante d'opérateurs est appelée de l'extérieur ou définie dans la mémoire rapide, éventuellement modifiée et exécutée d'après la macro-instruction.

En exécutant une notation dans le langage de programmation symbolique on ne tient pas compte des opérateurs commandant l'assemblage (des moyens de mise au point). Ils seront traités lors du travail de l'assembleur.

Exemple 6.22. Dans le langage de codage symbolique du calculateur « Minsk-32 », les notations se composant d'opérateurs d'action simples, s'exécutent à peu près de la même manière que les instructions de programmes établis dans le langage machine, avec cette différence qu'on utilise la mémoire « étiquette courante » au lieu du compteur ordinal. De plus, on suppose, pour les opérateurs d'action, que si le premier de deux opérateurs ou opérandes successifs d'une zone a l'adresse symbolique $\| N \|$, l'autre adresse symbolique sera donnée par l'expression $\| N + 1 \|$.

Si un programme dans le langage de codage symbolique occupe plus d'un bloc de mémoire, alors, pour organiser l'utilisation des adresses symboliques se rapportant à des blocs différents, il faut convenir que les contenus des registres de base sont tels que les blocs en question ne soient pas superposés.

Si l'opérateur courant est un appel de programme, alors d'abord sont exécutés les opérateurs correspondant à l'opérateur d'appel, puis les opérateurs suivant cet opérateur d'appel.

Notons que le langage de codage symbolique possède des opérateurs mettant l'exécution du programme compilé sous contrôle du système d'exploitation de la machine qui peut réaliser des interruptions de l'exécution du programme d'après les signaux spéciaux. Parmi les opérateurs de ce genre citons les opérateurs « attendre la réponse de l'opérateur (humain) »; « attendre » (la fin du travail d'un dispositif indiqué dans cet opérateur); « autoriser l'entrée par

interruption »; « interdire l'entrée par interruption »; « description d'entrée temps réel »; « description d'entrée par machine à écrire »; « impression sur la machine à écrire » (pour transmettre au calculateur un ordre de l'opérateur); « bloquer l'exécution du programme »; « arrêter l'exécution du programme »; « lire le temps »; « lire la date »; « répéter l'exécution de programme »; « charger un segment »; « libérer un segment ». Les opérateurs énumérés font suspendre l'exécution du programme jusqu'à un signal d'interruption (« attendre », « attendre la réponse de l'opérateur », « bloquer l'exécution du programme »), effectuent l'entrée et le traitement de l'information provenant de la source d'interruption (impression sur la machine à écrire », « autoriser l'entrée par interruption », « description d'entrée temps réel », « description d'entrée par machine à écrire », « lire le temps », « lire la date »), interdisent l'arrivée de l'information d'une source d'interruption (« interdire l'entrée par interruption »), ou bien arrêtent l'exécution du programme (« arrêter l'exécution du programme »), organisent la répétition d'une partie donnée de programme, assurent la mise en mémoire ou la sortie de la mémoire d'une partie de programme (« charger un segment », « libérer un segment »), etc.

Certains de ces opérateurs s'emploient dans des suites spéciales d'opérateurs d'action qui s'exécutent lorsqu'une interruption se produit les opérateurs (« interdire l'interruption », « autoriser l'interruption » sont mis respectivement au début et à la fin de la suite qui traite le signal d'interruption). D'autres opérateurs indiquent la disposition des opérateurs de cette suite (« description d'entrée temps réel », « description d'entrée par machine à écrire »).

En considérant l'exécution des notations dans le langage de codage symbolique, il faut supposer que l'automate qui l'organise réalise les mêmes opérations que le software du calculateur « Minsk-32». En particulier, il y a les opérations d'interruption, l'automate possède les cellules contenant l'heure et la date, il y a les moyens d'entrée (analogue à l'entrée par machine à écrire), de chargement des programmes, etc.

Exemple 6.23. L'exécution des algorithmes notés dans le langage d'assemblage est déterminée, dans une grande mesure aussi bien par le langage machine que par celui du système exécutif.

Les opérateurs d'action simples dans le langage d'assemblage s'exécutent dans l'ordre de notation, l'ordre d'exécution séquentiel n'étant dérangé que par les opérateurs de branchement et les macro-instructions. Le premier opérateur à exécuter est indiqué par l'opérateur désigné || START || dans la description du module de programme correspondant. Le passage d'un opérateur d'une section vers un opérateur d'une autre section se fait en utilisant l'information fournie par les opérateurs || EXTRN || et || ENTRY ||. L'automate qui

exécute une notation dans le langage d'assemblage assume les fonctions de l'éditeur des liens.

Au cours de l'exécution d'une macro-instruction, la macrodéfinition correspondante revêt une forme concrète et la suite obtenue d'opérateurs d'action s'exécute.

L'automate appelé à exécuter une notation doit agir d'une manière analogue au système opératoire en ce qui concerne la production des interruptions et les réactions sur ces interruptions.

Les opérateurs de description et d'allocation de mémoire s'interprètent de façon à éviter la superposition des sections de programme.

Les opérateurs de conversation sont ignorés par le compilateur.

§ 6.8. Caractéristique générale des langages de programmation symbolique du calculateur «Minsk-32» et du système IBM-360

Les langages considérés n'appartiennent, strictement parlant, à aucune des classes de langages indiquées au § 6.1. Ils ont des propriétés des langages de classes différentes.

Le langage de codage symbolique du calculateur « Minsk-32 » représente un autocode pour le langage du système exécutif, avec distribution automatique de la mémoire et expressions d'adresses.

Ce langage conserve les rangs non seulement des opérations essentielles, mais de toutes les opérations du système exécutif. Et pourtant il n'est pas un autocode 1 : 1 pour deux raisons ; premièrement, le langage admet les expressions d'adresses contenant des opérateurs qui sont interprétés par l'assembleur dans un sens déterminé (pour le calcul des valeurs d'adresses) ; deuxièmement, à une instruction de branchement dans le langage du système exécutif il correspond une collection d'opérateurs. Il n'y a pas non plus de correspondance complète entre les langages des opérandes du système exécutif et le langage de codage symbolique, car dans ce dernier on définit et utilise la construction « zone intérieure », absente dans le système exécutif.

Le langage de codage symbolique ne contient pas d'opérateurs qui commandent les organes extérieurs sans le système opératoire (exécutif), donc il doit nécessairement correspondre au langage de ce système.

Le langage d'assemblage IBM-360 représente la réunion de deux sous-langages dont l'un est un autocode pour le langage machine avec distribution automatique de la mémoire et expressions d'adresses, et l'autre, un autocode pour le langage du système exécutif avec distribution automatique de la mémoire et expressions d'adresses.

Considérons le premier sous-langage. Il conserve les rangs non seulement des opérations essentielles, mais de toutes les opérations

de langage machine. Il n'est pas un autocode 1 : 1 pour le langage machine pour les mêmes raisons que ne l'est pas le langage de programmation symbolique de la machine « Minsk-32 ». En même temps, ce sous-langage correspond au langage machine, parce qu'il y existe des opérateurs correspondant à toute instruction du langage machine.

Le second sous-langage du langage d'assemblage conserve les rangs de toutes les opérations du langage du système exécutif. N'étant pas un autocode 1 : 1 par rapport au langage du système exécutif pour les raisons indiquées plus haut, il lui correspond, car on peut y utiliser toutes les macro-instructions définies dans le langage du système exécutif.

CHAPITRE 7

INTRODUCTION AU LANGAGE ALGOL-60

En 1960, un groupe de savants de différents pays élabora le langage algorithmique ALGOL-60. Selon l'idée des auteurs, il fut destiné à devenir un langage international de programmation automatique de problèmes scientifiques et un moyen de faciliter l'échange d'algorithmes. En élaborant l'ALGOL-60 on visait trois objectifs principaux:

a) d'obtenir un langage qui soit autant proche que possible du langage usuel des descriptions mathématiques;

b) d'assurer la possibilité de donner en ce langage les algorithmes de résolution des problèmes de calcul numérique;

c) de rendre aisée la traduction automatique des algorithmes donnés en ce langage en les programmes pour des ordinateurs concrets.

Conformément à ces conditions, on proposa trois types de versions du langage ALGOL. L'une des versions fut déclarée *langage étalon*. Il contient un ensemble fixe de symboles de base et ne diffère des autres versions que par la forme de ces symboles.

Un groupe de versions s'appellent *langages de publications*. Dans un langage de publications il est permis d'utiliser n'importe quels signes en tant que symboles de base, à condition d'observer une correspondance exacte avec le langage étalon, c'est-à-dire qu'il soit établi à quoi exactement, dans le langage étalon, correspond chaque signe ou façon de notation. Entre tous les symboles de base, sauf les lettres *), du langage étalon et d'un langage de publications, il doit exister une correspondance biunivoque. Quant aux lettres du langage de publications, il suffit qu'on connaisse les signes qui sont ses lettres.

Il y a, enfin, encore un groupe de versions du langage ALGOL qui s'appellent *représentations concrètes*. Ces versions diffèrent du langage étalon et des langages de publications par ce qu'aux symboles de base du langage étalon il correspond dans une représentation concrète les signes ou les combinaisons de signes admis dans cette

---

*) Le sens du terme « lettre » ne coïncide pas avec son sens dans la théorie des algorithmes et sera défini plus bas.

représentation et qui sont des signes de l'alphabet d'entrée d'un calculateur concret.

En donnant des exemples de notations dans le langage ALGOL nous n'avons pas à nous servir des signes de ponctuation, puisqu'ils font partie des symboles principaux du langage ALGOL même. Pour cette raison, dans le cas où il faudra séparer une notation en ALGOL d'une autre ou du texte français, nous nous servirons, au lieu de signes de ponctuation (points, virgules, guillemets), du double trait vertical || qui n'est pas un symbole de base de l'ALGOL.

En décrivant l'ALGOL nous tenons à ce que l'exposé soit parfaitement rigoureux et en même temps le plus confortable possible. Pour cette raison, aucun métalangage formel ne sera plus utilisé.

La sémantique de l'ALGOL est décrite, premièrement, par l'explication du sens de chaque type d'instruction et, deuxièmement, au moyen de ce qu'on appelle algorithme d'exécution d'un programme ALGOL (§ 7.9).

Un algorithme donné en ALGOL s'appelle *programme ALGOL*. Ses constructeurs le considèrent comme un langage opératoriel. En lisant les descriptions de l'« action » des instructions complexes il faut avoir en vue que dans un programme ALGOL les instructions s'exécutent dans l'ordre de leur notation, sauf si l'effet d'une instruction consiste précisément en une modification de l'ordre séquentiel d'exécution.

§ 7.1. Alphabet de l'ALGOL

Les symboles de base du langage ALGOL qui forment son alphabet sont :

— les *chiffres* arabes : || 0 || 1 || 2 || 3 || 4 || 5 || 6 || 7 || 8 || 9 || ;

— les *lettres* majuscules et minuscules italiques de l'alphabet latin ;

— les *valeurs logiques* || **true** || et || **false** || (chacun de ces mots représente l'unique symbole, ce qui est interprété dans la présente description du langage par les caractères demi-gras) ;

— les *délimiteurs* qui sont partagés en signes d'opérations, séparateurs et descripteurs.

Les *signes d'opérations* sont :

— les signes d'opérations arithmétiques || + || — || × || / || ÷ || ↑ || (addition, soustraction, multiplication, division, division sans reste, exponentiation) ;

— les signes des relations || < || ≤ || > || ≥ || = || ≠ || qu'on lit respectivement « est inférieur à », « est inférieur ou égal à », « est supérieur à », « est supérieur ou égal à », « est égal à », « n'est pas égal à » ;

— les signes d'opérations logiques || ∧ || ∨ || ¬ || ⊃ || ∼ || qu'on lit respectivement « et », « ou », « non », « implique », « équivaut ».

Les *séparateurs* sont :

— les séparateurs constructifs || , || . || $_{10}$ || : || ; || ⊔ || := || qui s'appellent respectivement virgule, point, dix, deux points, point virgule, blanc, signe d'affectation ;

— les parenthèses || (||) ||, les crochets || [||] ||, les parenthèses d'instruction || **begin** || **end** ||, les guillemets || ‘ || ’ ||, tous ces symboles existant par couples (un symbole ouvrant, un symbole fermant). Les symboles || **begin** || et || **end** || sont indivisibles, ce qui est exprimé par les caractères demi-gras ;

— les opérateurs séquentiels || **go to** || **if** || **then** || **else** || **for** || **do** || **step** || **until** || **while** || qui sont des symboles indivisibles.

Les *descripteurs* sont || **integer** || **real** || **Boolean** || **array** || || **switch** || **procedure** || **own** || **value** || **label** || **string** || **comment** ||, ils représentent des symboles indivisibles.

Les structures primaires de l'ALGOL qui, à côté de certains symboles isolés, peuvent faire partie de structures plus compliquées sont les nombres, les identificateurs et les chaînes. Décrivons-les plus en détail.

§ 7.2. Structures primaires

Nombres. Chaque suite finie de chiffres s'appelle nombre entier sans signe (en abrégé, *entier sans signe*). Un entier sans signe, ainsi qu'un entier sans signe précédé du signe || + || ou || — || s'appelle *entier*.

Exemple 7.1. Voici quelques entiers sans signe :

|| 000 || 00 || 133 || 012 || 5 ||.

Voici des exemples d'entiers :

|| —00 || —295 || +527 || +0 || 00 || 133 || 012 ||.

On appelle *fraction régulière* un entier sans signe précédé d'un point. On appelle *nombre décimal* un entier sans signe, ou une fraction régulière, ou encore un entier sans signe suivi d'une fraction régulière.

Exemple 7.2. Voici des fractions régulières :

|| .000 || .17 || .017 ||

et des nombres décimaux :

|| 135 || .58 || 145.19 || 0.293 ||.

On appelle *exposant* (décimal) le séparateur $\|_{10}\|$ suivi d'un entier (sans ou avec signe).

EXEMPLE 7.3. Les notations suivantes sont des exposants:

$$\|\,_{10}15\,\|_{10}-27\,\|_{10}+017\,\|.$$

On appelle *nombre sans signe* un nombre décimal, un exposant ou un nombre décimal suivi d'un exposant.

EXEMPLE 7.4. Voici quelques nombres sans signe qui ne sont pas nombres décimaux ni exposants:

$$\|\,15_{10}-8\,\|\,8.75_{10}+32\,\|\,0.07_{10}-03\,\|.$$

On appelle *nombre* un nombre sans signe ou une chaîne se composant d'un signe $\|+\|$ ou $\|-\|$ et d'un nombre sans signe.

Les nombres de l'ALGOL représentent une forme de notation des nombres au sens ordinaire. A la virgule utilisée dans les notations ordinaires des nombres il correspond le point dans les notations de l'ALGOL (c'est l'écriture anglo-saxone qui a été adoptée). Les nombres ALGOL qui contiennent le séparateur dix correspondent à la forme normale de notation des nombres dans le système de numération décimal.

EXEMPLE 7.5. Les nombres ALGOL des exemples 7.3 et 7.4 sont dans la notation habituelle 10^{15}; 10^{-27}; 10^{17}; $15\cdot10^{-8}$; $8{,}75\cdot10^{32}$; $0{,}07\cdot10^{-3}$.

Identificateurs. On appelle identificateur une suite finie de lettres et de chiffres qui commence par une lettre.

EXEMPLE 7.6. Voici des identificateurs:

$$\|\,x\,\|\,A\,\|\,Canal\,\|\,x2\,\|\,ASB\,\|\,Alpha\,\|.$$

Les identificateurs s'utilisent dans l'ALGOL en tant que composantes principales des notations des noms de grandeurs, de fonctions, etc. Certains noms se réduisent tout simplement à des identificateurs.

Chaînes. Une suite de symboles de base sans guillemets s'appelle *corps de chaîne*. En particulier, le corps de chaîne peut être représenté par une suite de symboles vide. On appelle *chaîne* un corps de chaîne mis entre guillemets, ou bien une suite se composant de chaînes et de corps de chaînes et mise entre guillemets.

EXEMPLE 7.7. Les notations $\|\,a$, **if** $+\,\|\,a+b\,\|$ *calculer* $x1\,\|$ sont des corps de chaînes.

Les notations ||‚$a + b$'||‚'||‚[[1 ‚$x + y$'‚'2x'||‚*une chaîne* ⊔ *se désigne* ⊔ *par le symbole* ⊔ ‚*string*' ⊔' || sont des chaînes.

Notons que le symbole || ⊔ || (blanc) est ignoré dans l'ALGOL s'il se trouve à l'extérieur de chaînes. Les chaînes de l'ALGOL ne s'utilisent qu'en tant que paramètres effectifs dans les instructions de procédures.

§ 7.3. Variables. Tableaux. Descriptions de type

Le terme *variable simple* désigne une quantité qui prend des valeurs numériques ou logiques et qui est désignée par un identificateur.

On appelle *tableau* un ensemble fini des quantités qui sont désignées par un même identificateur et correspondent aux collections ordonnées de valeurs de certains paramètres entiers dont chacun prend toutes ses valeurs, à partir de la plus petite jusqu'à la plus grande.

Lesdites valeurs minimale et maximale sont appelées *paire de bornes* du paramètre correspondant. L'identificateur commun de toutes les quantités faisant partie d'un tableau s'appelle *identificateur de tableau*. Un tableau est désigné par un identificateur suivi d'une liste de paires de bornes écrite entre crochets. Les paires de bornes de cette notation sont séparées l'une de l'autre par des virgules, les valeurs minimale et maximale (de chaque paire) étant séparées par le signe ||:||. Dans le cas général, les éléments des paires sont représentés par des expressions arithmétiques (voir § 7.6) qui s'appellent alors *expressions en indice*. Le nombre de paires de bornes d'un tableau est sa *dimension*.

Pour désigner un élément de tableau, on se sert d'une *variable indicée* qui s'écrit comme l'identificateur de tableau suivi d'une liste d'indices (séparés par des virgules) mise entre crochets. Le nombre d'indices dans la liste doit être égal à la dimension du tableau. Les indices peuvent s'exprimer par des valeurs de paramètres ou par des expressions arithmétiques (qu'on appelle alors expression en indice).

EXEMPLE 7.8. On peut désigner les variables simples comme

|| x || $x2$ || *Température* || *Poids* 1 || *Poids* 2 ||

et les tableaux comme

|| $a\,[0:3]$ || *Table* $[1:6,\ 4:7]$ || $u\,[k:k+3,\ l:l\uparrow 2]$ ||.

Pour les tableaux ci-dessus, les variables indicées peuvent avoir la forme

|| $a\,[2]$ || $a\,[i]$ || *Table* $[i,\ j]$ || $u\,[j+1,\ i+j]$ ||.

Les notions de variable simple et de variable indicée sont réunies en la notion de *variable*.

Il y a trois types de variables: variables *réelles*, *entières* et *logiques*. Chaque variable ne peut prendre que des valeurs correspondant à son type. Pour désigner les types de variables on utilise respectivement les symboles de base || **real** || **integer** || **Boolean** ||.

Le type de variables simples est donné par ce qu'on appelle *description du type* qui se compose d'un nom de type suivi d'une liste de variables simples. Le nom du type s'exprime par un descripteur ou un couple de descripteurs:
|| **integer** || **real** || **Boolean** || **own integer** || **own real** || **own Boolean** ||.

La liste de variables simples représente une suite d'identificateurs séparés par des virgules. La signification du descripteur || **own** || est expliquée au p. 7.7.2.

Exemple 7.9. Les notations suivantes sont des descriptions du type:
|| **integer** x, y, z, *Numéro*, $u7$ || **own real** u, v, *Poids* ||
Boolean $p1$, $p2$, *Condition* 1 ||.

Le type de variables indicées est donné par ce qu'on appelle *description de tableau* qui se compose d'un nom de type (décrit plus haut), d'un descripteur || **array** || et d'un nombre de segments séparés par des virgules. Chaque segment représente une succession d'identificateurs de tableaux séparés par des virgules, le dernier identificateur étant suivi d'une liste de paires de bornes mise entre crochets. Si le nom du type se réduit à l'unique descripteur || **real** ||, il peut être omis.

Exemple 7.10. Voici quelques descriptions de tableaux:
|| **integer array** A, B [1 : 6, 3 : 50, 5 : 120], u, v, w [1 : 20] ||
Boolean array *Valeur* [1 : 5, 1 : 12] || **array** w [1 : 100] ||
own real array x [1 : 80] ||.

Dans tout segment, la liste de paires de bornes se rapporte à tous les identificateurs qui la précèdent (et qui n'en sont pas séparés par d'autres listes). Ainsi, une description de tableau peut fournir l'information soit sur un seul, soit sur plusieurs tableaux à la fois. Un nom du type se rapporte à tous les tableaux de la description.

Remarquons que les variables figurant dans les expressions qui sont contenues dans les paires de bornes doivent être décrites dans un bloc qui englobe le bloc contenant la description du tableau. Il ne faut pas l'oublier en lisant le p. 7.7.2.

§ 7.4. Langage des opérandes lié à l'ALGOL

Les auteurs de l'ALGOL ne disent rien sur le langage des opérandes lié à ce langage algorithmique. Quant à la description de l'ALGOL elle ne permet de juger que sur quelques particularités du langage des opérandes et contient, en réalité, des données sur la classe des langages pouvant servir de langages des opérandes. En concrétisant certaines notions sur les opérandes, on peut décrire le langage des opérandes de l'ALGOL comme suit.

L'alphabet du langage des opérandes contient tous les symboles de base de l'alphabet de l'ALGOL et, de plus *), le symbole $\| \equiv \|$ qui se lit « contient », le séparateur $\| \nabla \|$ et le séparateur $\|_0\|$. Les structures primaires de l'ALGOL (nombre et identificateur) sont aussi les structures primaires du langage des opérandes.

Les opérandes d'un algorithme donné en ALGOL représentent ce qu'on appelle *états de mémoire*. Un état de mémoire est la notation **) se composant de deux parties : d'un *état de mémoire extérieure* et d'un *état de mémoire intérieure*.

On appelle état de mémoire extérieure une notation, vide ou formée par des éléments d'état de mémoire extérieure appelés *canaux*. Un canal représente une notation se composant d'un identificateur $\|$*canal*$\|$, d'un entier sans signe, du symbole $\| \equiv \|$ et d'une suite dont les éléments peuvent être des nombres, des entiers et des symboles de base de l'ALGOL. Les éléments de la suite sont séparés l'un de l'autre par des séparateurs $\| \nabla \|$, la fin de la suite étant désignée par un séparateur $\|_0\|$. L'entier sans signe qui suit l'identificateur $\|$*canal*$\|$ s'appelle *numéro du canal*. Les numéros des canaux qui font partie de l'état de mémoire extérieure doivent être différents entre eux. La notation qui suit le symbole $\| \equiv \|$ s'appelle *contenu du canal*.

Exemple 7.11. Un état de mémoire extérieure peut être représenté par

$$\| \mathit{canal}\ 1 \equiv +2_{10}03\nabla\ \mathbf{true}\ \nabla+25.10\nabla-25_0\ \mathit{canal}\ 2 \equiv$$
$$\equiv 25_0\ \mathit{canal}\ 3 \equiv x\nabla 1\nabla := \nabla 25.3_{10}5_0 \|.$$

Un état de mémoire intérieure peut représenter une notation vide (alors on ne l'écrit pas), ou une suite de notations appelées *éléments d'état de mémoire intérieure*. Un élément d'état de mémoire intérieure est désigné par une notation qui commence par un entier sans signe (appelé *niveau* de l'élément), suivi d'un nom de variable, suivi d'un séparateur $\| \equiv \|$, suivi d'une valeur de la variable et finissant par un séparateur $\|_0\|$. Le nom de variable peut être donné par un identificateur (si la variable est simple), ou par un identificateur

*) Les symboles $\| \equiv \| \nabla_0 \|$ sont introduits par les auteurs.
**) Cette structure des opérandes est l'une des possibles.

suivi d'une liste, entre crochets, des indices séparés par des virgules (si la variable est indicée).

Exemple 7.12. L'état de mémoire intérieure peut avoir la forme

$$\| 1x \equiv +5_{10} - 3_0 2x \equiv \mathbf{true}_0\ 1y \equiv -275_0 1u\ [1,\ 2,\ 1] \equiv 5_0 \|.$$

Un exemple d'état de mémoire sera obtenu si l'on fait suivre la notation de l'exemple 7.11 de la notation du présent exemple. La notation de l'exemple 7.11 peut être un état de mémoire correspondant à l'état de mémoire intérieure vide.

§ 7.5. Indicateurs des fonctions

En plus des signes d'opérations qui sont des symboles de base on utilise dans l'ALGOL des signes d'opérations plus compliqués appelés indicateurs de fonctions. On exécute les opérations définies par un indicateur de fonction en exécutant l'algorithme correspondant donné au moyen des « déclarations de procédure » (voir p. 7.7.8). Les notations définissant les valeurs des arguments d'une fonction sont appelées paramètres effectifs. Les paramètres effectifs peuvent être des chaînes, des identificateurs de tableaux, des identificateurs de procédures, des identificateurs d'aiguillage et des expressions (voir §§ 7.2, 7.3, 7.6, pp. 7.7.5, 7.7.6).

Un indicateur de fonction s'écrit comme un identificateur suivi d'une liste, entre parenthèses, de paramètres effectifs séparés par des délimiteurs de paramètres qui sont soit des virgules, soit des notations de la forme

$$\|) \ldots : (\|,$$

où les points désignent une suite quelconque de lettres. Voici des exemples d'indicateurs de fonctions:

|| *r* || *f* (*x*, *y*) || *F* (3, *t*, *p*) || *F* (3) *température*: (*t*) *pression*: (*p*) ||.

Dans le premier indicateur, la liste des paramètres effectifs n'est pas présente ou, comme on dit, est vide. Le dernier indicateur contient les mots expliquant le sens de ses paramètres effectifs.

La valeur d'une fonction peut représenter ou bien un nombre réel, ou bien un nombre entier, ou enfin une valeur logique, selon son type (qui est indiqué dans la déclaration de procédure correspondant à la fonction).

Les fonctions standard sont désignées dans l'ALGOL par des identificateurs spéciaux qui leur sont réservés:

|| *abs* (*A*) || valeur absolue de l'expression *A*,

|| *sqrt* (*A*) || racine carrée de l'expression *A*,

|| *sin* (*A*) || *cos* (*A*) || *tg* (*A*) || *ctg* (*A*) || fonctions trigonométriques,

|| *arcsin* (*A*) || *arccos* (*A*) || branches principales des fonctions trigonométriques inverses,

|| *ln* (*A*) || logarithme naturel de l'expression *A*,

|| *entier* (*A*) || le plus grand entier qui ne dépasse pas la valeur de l'expression *A* (l'argument est du type **real**, le résultat, du type **integer**),

|| *sign* (*A*) || fonction qui vaut +1 si $A > 0$, —1 si $A < 0$ et 0 si $A = 0$,

|| *exp* (*A*) || fonction e^A.

Chacune de ces fonctions est définie pour les arguments du type **real** aussi bien que du type **integer**. Les valeurs des fonctions *entier* et *sign* sont du type **integer**, celles des autres fonctions, du type **real**.

§ 7.6. Expressions dans l'ALGOL

7.6.1. Expressions logiques. La définition de l'expression logique (ou booléenne) est récursive.

On appelle *primaires logiques* les valeurs logiques, les variables logiques, les indicateurs de fonctions logiques, les relations et, enfin, les expressions logiques mises entre parenthèses.

On appelle *relation* un couple d'expressions arithmétiques simples (voir p. 7.6.2), en particulier de nombres et d'identificateurs de variables numériques, liés par un signe d'opération de relation. Voici des exemples de relations:

|| $x > 1$ || $sin\ (x) \leqslant 0.5$ || $2 \times$ *vitesse* $\neq$ *accélération* ||.

On appelle *secondaire logique* un primaire logique précédé ou non du signe || $\neg$ ||.

On appelle *facteur logique* un secondaire logique, ainsi qu'une suite constituée par un facteur logique, le signe || $\wedge$ || (et) et un secondaire logique. Autrement dit, un facteur logique représente une suite finie de secondaires logiques liés par les symboles || $\wedge$ ||.

On appelle *terme logique* un facteur logique isolé, ainsi que toute suite se composant d'un terme logique, d'un signe || $\vee$ || (ou) et d'un facteur logique. Un terme logique représente donc une suite finie de facteurs logiques liés par les signes || $\vee$ ||.

On appelle *implication* un terme logique, ainsi que toute suite se composant d'une implication, d'un signe || $\supset$ || et d'un terme logique (le signe $\supset$ se lit « implique »).

On appelle *expression logique simple* une implication, ainsi que toute suite se composant d'une expression logique simple, d'un signe || $\supset$ || et d'une implication. La notion d'expression logique simple réunit les notions de primaire logique, de secondaire logique, de facteur logique, de terme logique et d'implication.

On appelle *condition* une suite se composant d'un symbole || **if** ||, d'une expression logique (arbitraire) et d'un symbole || **then** ||. Voici des exemples des conditions:

|| **if** $x > 0$ **then** || **if** $x > 0 \wedge y < 2$ **then** || **if** $x > 0 \supset y < 0$ **then** ||.

On appelle *expression logique* une expression logique simple, ainsi que toute suite se composant d'une condition, d'une expression logique simple, d'un symbole || **else** || et d'une expression logique (arbitraire).

Exemple 7.13. Les notations suivantes sont des expressions logiques :

|| **if** $x > 0$ **then** $x + 1 > 10$ **else** $x + 1 \leqslant y$ || **if** $x > 0$ **then** $y > 1$ **else if** $y > 1$ **then** $z \geqslant 3$ **else** $z \geqslant 8$ || **if** $x \geqslant 2$ **then** (**if** $y > 1$ **then** $z \geqslant 3$ **else** $z \leqslant 8$) **else** $y > 1$ || $x \neq y \wedge y = z$ || $p1 \wedge p2 \vee p1 \wedge \neg\, p2$ || **false** ||, etc.

7.6.2. Expressions arithmétiques. La définition de l'expression arithmétique est récursive, sa construction est analogue à celle de la définition de l'expression logique.

On appelle *primaire arithmétique* un nombre, une variable, un indicateur de fonction ou une expression arithmétique (arbitraire) mise entre parenthèses.

On appelle *facteur* soit un primaire arithmétique, soit une suite se composant d'un facteur, d'un signe || $\uparrow$ || et d'un primaire.

On appelle *terme* un facteur isolé, ainsi que toute suite se composant d'un terme, de l'un des signes || $\times$ ||, || / ||, || $\div$ || et d'un facteur.

On appelle *expression arithmétique simple* un terme isolé précédé ou non de l'un des signes || + || et || — ||, ainsi que toute suite se composant d'une expression arithmétique simple, de l'un des signes || + || et || — || et d'un terme.

On appelle *expression arithmétique* une expression arithmétique simple isolée, ainsi que toute suite se composant d'une condition (v. p. 7.6.1), d'une expression arithmétique simple, d'un symbole || **else** || et d'une expression arithmétique (arbitraire).

Le sens des signes des opérations arithmétiques dans l'ALGOL correspond à leur sens usuel. Le sens du signe || $\uparrow$ || est bien interprété par son appellation « exponentiation ». C'est le signe || $\div$ || qui demande une explication.

L'opération « division sans reste » est définie dans le langage mathématique usuel comme calcul de la partie entière de la valeur absolue du quotient prise avec le signe du quotient, i.e.

$$a \div b = \operatorname{sign} \frac{a}{b} E \left| \frac{a}{b} \right| .$$

EXEMPLE 7.14. Les notations suivantes sont des expressions arithmétiques:

|| −25.456 || **if** $x > 0$ **then** x **else** 1 || **if** $x > y$ **then** $(x + y)$ **else** $(x - y) \uparrow 2 + 1$ || $2 +$ (**if** $x > 3$ **then** $2 + y$ **else** 3) $+ 3 \times x$ ||.

EXEMPLE 7.15. Les valeurs de la fonction qui, en langage mathématique traditionnel, est donnée par la formule

$$y = \begin{cases} x+1 & \text{pour } x < 0, \\ 10 & \text{pour } x = 0, \\ 2-x & \text{pour } x > 0, \end{cases}$$

peuvent être données à l'aide de l'expression arithmétique

|| **if** $x < 0$ **then** $x + 1$ **else if** $x = 0$ **then** 10 **else** $2 - x$ ||.

Une expression arithmétique qui sert à calculer les valeurs d'un paramètre numérique entier (d'un indice) s'appelle *expression en indice*. Une expression en indice ne doit prendre que des valeurs entières non négatives. Si tout de même le calcul fournit une valeur réelle (non entière), alors on choisit pour valeur de l'expression || *entier* $(A + 0.5$ ||, où A est le résultat du calcul, et l'opération || *entier* || a le sens décrit au § 7.5.

7.6.3. Expressions de désignation. Les instructions ALGOL (§ 7.7) qui font partie d'un algorithme peuvent être repérées, i.e. pourvues de noms individuels. Ces noms d'instructions s'appellent *étiquettes*. Une étiquette est soit un identificateur soit un entier sans signe. Les expressions de désignation servent à déterminer des étiquettes.

On appelle *expression de désignation simple* une étiquette, un indicateur d'aiguillage ou toute expression de désignation mise entre parenthèses.

On appelle *indicateur d'aiguillage* (p. 7.7.5) toute suite se composant d'un identificateur d'aiguillage et d'une expression d'indice arbitraire mise entre crochets (i.e. d'une expression arithmétique destinée au calcul d'indices).

On appelle *expression de désignation* soit une expression de désignation simple soit une suite se composant d'une condition (p. 7.6.1), d'une expression de désignation simple, d'un symbole || **else** || et d'une expression de désignation arbitraire.

EXEMPLE 7.15. L'étiquette || *calcul* || et l'indicateur d'aiguillage || *passage* $[i + j]$ || sont des expressions de désignation simples (donc des expressions de désignation).

Les notations

|| **if** $i \leqslant 2$ **then** *calcul* **else if** $i \leqslant 5$
then *passage* [i] **else** *passage* [$i + j$] ||

et

|| **if** $i > 2$ **then** (**if** $i \leqslant 5$ **then** *passage* [i]
else *passage* [$i + j$] *else* **calcul** ||

sont des expressions de désignation composées.

7.6.4. Calcul des expressions. Considérons une expression de la forme

|| **if** P **then** W_1 **else** W_2 ||,

où P est une expression logique, W_1 une expression logique *simple*, ou une expression arithmétique, ou une expression de désignation, W_2 est une *expression* du même type que W_1. Le calcul de W_3 se fait de la manière suivante. On calcule d'abord la valeur de P. Si l'on obtient la valeur || **true** ||, on considère comme valeur de W_3 celle de l'expression simple W_1. C'est elle donc qui est à calculer. Et si le calcul de P amène au || **false** ||, alors on considère comme valeur de W_3 celle de W_2.

Lorsqu'on a à calculer la valeur d'une expression W qui ne commence pas par un symbole || **if** ||, on examinera l'expression W de gauche à droite jusqu'à ce qu'on tombe sur la première expression simple calculable, qui se compose d'un signe d'opération et des opérandes. Une expression simple est dite calculable, si

a) les quantités qu'elle contient ont déjà des valeurs déterminées, c'est-à-dire qu'elles sont soit des variables données avant le calcul de l'expression, soit sont obtenues au cours du calcul;

b) ces quantités ne sont pas des données initiales d'opérations prioritaires qui ne sont pas encore exécutées.

Si la succession d'opérations arithmétiques n'est pas déterminée par des parenthèses, on admet qu'elles s'exécutent dans l'ordre de priorité suivant (ordre croissant):

1) || + || − ||,

2) || × || / || ÷ ||,

3) || ↑ ||.

Il existe donc trois niveaux de priorité pour les opérations arithmétiques. Les opérations de même priorité s'effectuent toujours dans l'ordre de leur notation de la gauche vers la droite. Pour les opérations logiques, il existe sept degrés de priorité (par ordre de

croissance) :

1) $\| \sim \|$,

2) $\| \supset \|$,

3) $\| \vee \|$,

4) $\| \wedge \|$,

5) $\| \neg \|$,

6) $\| < \| \leqslant \| > \| \geqslant \| = \| \neq \|$,

7) les opérations arithmétiques dont l'ordre de priorité est déjà vu.

Une fois la valeur d'une expression calculable simple trouvée, on considère celle-ci comme remplacée par cette valeur. On poursuit le calcul en itérant le procédé décrit jusqu'à obtenir la valeur cherchée de l'expression W.

Si lors du calcul d'une expression on rencontre un indicateur de fonction dont les paramètres effectifs sont déjà obtenus, on doit interrompre le calcul de l'expression, appeler la description de la procédure-fonction qui répond à l'indicateur en question, déterminer en exécutant la procédure la valeur de la fonction, revenir à l'expression calculée et considérer dorénavant la valeur obtenue de la fonction comme valeur de son indicateur (dans l'expression). Ceci fait, on poursuit le calcul de la valeur de l'expression.

Exemple 7.16. L'expression arithmétique

$$\| x + y \times z/u \uparrow 2 + f\,(x + (\textbf{if } x > 2 \textbf{ then } (y + 3) \textbf{ else } z - 1)) \|$$

est calculée de la façon suivante :

a) on calcule $\| y \times z \|$, on obtient une valeur r_1;

b) on calcule $\| u \uparrow 2 \|$, on obtient r_2;

c) on calcule $\| r_1/r_2 \|$, on obtient r_3;

d) on calcule $\| x + r_3 \|$, on obtient r_4;

e) on calcule $\| x > 2 \|$; supposons qu'on obtienne $\| \textbf{true} \|$;

f) on calcule $\| y + 3 \|$ et on obtient r_5 qui est considéré comme valeur de toute l'expression primaire $\| (\textbf{if } x > 2 \textbf{ then } (y + 3) \textbf{ else } z - 1) \|$;

g) on calcule $\| x + r_5 \|$, on obtient r_6;

h) on calcule $\| f\,(r_6) \|$ en s'adressant à la procédure-fonction; on obtient r_7;

i) on calcule $\| r_4 + r_7 \|$. On obtient le résultat qui représente la valeur de l'expression W.

§ 7.7. Instructions

Le langage algorithmique ALGOL-60 est, à l'avis de ses constructeurs, un langage opératoriel (on verra au § 7.10 si c'est vrai). Les instructions (qui sont opérateurs au sens des chapitres précédents) sont des éléments principaux de la structure de l'ALGOL au moyen desquels on écrit les algorithmes (appelés également programmes ALGOL). Les expressions décrites plus haut font partie d'instructions. Une instruction peut être *non étiquetée* ou *étiquetée* par une ou plusieurs étiquettes. Si N désigne une instruction non étiquetée, et M_1, M_2, . . ., M_n sont des étiquettes, alors les notations $\| N \|$ $M_1 : N \|$ $M_1 : M_2 : N \|$ . . . $\| M_1 : M_2 : \ldots : M_n : N \|$ sont des instructions (ici N et M_1, M_2, . . ., M_n sont des métasymboles. et le signe $\| : \|$ est un symbole de l'ALGOL). On utilise comme étiquettes soit des identificateurs, soit des entiers sans signe. Les instructions de l'ALGOL (voir table 7.1) sont *inconditionnelles* et *conditionnelles*. Parmi les instructions inconditionnelles il y a des instructions *de base* (simples) et des instructions composées. Il existe quatre types d'instructions de base : *instructions d'affectation*, *instructions de branchement*, *instructions vides* et *procédures*. Il y a deux types d'instructions inconditionnelles composées : *instructions composées* et *blocs*.

Table 7.1

Classification des instructions de l'ALGOL

<table>
<tr><th colspan="11">Instructions de l'ALGOL</th></tr>
<tr><th colspan="9">Instructions exécutables</th><th colspan="2" rowspan="2">Instructions non exécutables</th></tr>
<tr><th colspan="6">Instructions inconditionnelles</th><th colspan="3">Instructions conditionnelles</th></tr>
<tr><th colspan="4">Instructions simples (de base)</th><th colspan="2">Instructions composées</th><th colspan="3">Instructions composées</th><th>Instructions simples</th><th>Instructions composées</th></tr>
<tr><td>Instruction d'affectation</td><td>Instruction de branchement</td><td>Instruction procédure</td><td>Instruction vide</td><td>Instruction composée</td><td>Bloc</td><td>Boucle</td><td>Instruction si</td><td>Instruction alternative</td><td>Aiguillage</td><td>Procédure</td></tr>
</table>

Les istructions conditionnelles sont toutes composées (il n'y a pas d'instructions conditionnelles simples). Il existe trois types

d'instructions conditionnelles: *boucles, instructions* **si** et *instructions alternatives.*

De plus, il faut considérer comme instructions les *aiguillages* et les *procédures* qui sont pourtant des instructions non exécutables apparaissant dans les algorithmes sous forme de descriptions. Il ne faut pas confondre une procédure avec une instruction de procédure qui réalise, comme on le verra plus bas, l'appel de la procédure.

Voyons les instructions de chaque type.

7.7.1. Instruction composée. La forme de l'instruction composée est

$$\| \textbf{begin } S_1 ; \ S_2 ; \ \ldots ; \ S_n \textbf{ end} \|,$$

où S_1, S_2, . . ., S_n sont des instructions quelconques.

Ainsi, une instruction composée est une suite d'instructions placées entre les symboles **begin** et **end** et séparées par des signes $\|;\|$.

Exécuter une instruction composée c'est exécuter les instructions qui en font partie dans l'ordre de leur notation (sauf si l'exécution de quelques-unes d'entre elles consiste en la modification de l'ordre de leur exécution).

7.7.2. Bloc. Un b*loc* est une notation de la forme

$$\| \textbf{begin } B_1 ; \ B_2 ; \ \ldots ; \ B_m ; \ S_1 ; \ S_2 ; \ \ldots ; \ S_n \textbf{ end} \|,$$

où S_i sont toujours des instructions, B_1, B_2, . . ., B_m des déclarations. On distingue quatre sortes de déclarations: celles de type, de tableau, d'aiguillage et de procédure.

Ainsi, un bloc est une suite, mise entre les symboles **begin** et **end**, d'instructions et de déclarations séparées par des symboles $\|;\|$. Les déclarations précèdent les instructions.

Les variables déclarées dans un bloc sont dites *locales* pour ce bloc. Les variables qui ont un sens en dehors d'un bloc donné sont dites *non locales* pour lui. Habituellement, les variables non locales sont déclarées quelque part dans un bloc contenant le bloc donné.

L'exécution d'un bloc commence par ce que toutes les variables qui y sont déclarées acquièrent une nouvelle signification (on leur fait correspondre les éléments d'état de la mémoire interne dont les niveaux dépassent d'une unité les niveaux des éléments qui correspondaient à ces variables en dehors du bloc donné). Ceci fait, les instruction contenues à l'intérieur du bloc s'exécutent dans l'ordre de leur notation (sauf s'il s'agit de modifier l'ordre de leur exécution). Avant la « sortie » du bloc, les valeurs de toutes les variables qui y sont déclarées, à l'exception de celles déclarées à l'aide d'un déclarateur $\|$ **own** $\|$, sont complètement « oubliées ».

Les valeurs des variables du type || **own** || sont retenues jusqu'au moment d'une nouvelle « entrée » dans le bloc. Après la sortie du bloc, les variables qui y sont déclarées reprennent leur signification initiale.

Remarquons que les variables propres du bloc, i.e. décrites au moyen d'un déclarateur || **own** ||, se comportent comme les variables non locales, déclarées dans un bloc qui contient le bloc donné. Les variables indicées qui correspondent à des tableaux propres (|| **own** ||) à bornes variables présentent une exception. On ne peut réduire de telles variables à des variables non locales. Le comportement des variables propres au cours de l'exécution d'un programme ALGOL est décrit au § 7.9. Dans certaines versions de l'ALGOL utilisées en pratique, l'emploi des tableaux propres à bornes variables n'est pas autorisé ce qui présente une limitation considérable.

En construisant des blocs contenant des déclarations de tableaux à bornes variables, il ne faut pas oublier que les variables et les indicateurs de fonctions qui figurent dans les expressions des bornes doivent être déclarés non pas dans le bloc donné, mais dans un bloc comprenant le bloc donné. Les valeurs des bornes inférieures et supérieures des tableaux déclarés dans le bloc sont calculées une fois pour chaque entrée dans le bloc. Si les indices de la variable correspondant à un tableau dépassent ces bornes, le processus algorithmique s'arrête sans résultat (même si le tableau est du type || **own** ||).

7.7.3. Instruction d'affectation. On appelle *instruction d'affectation* une instruction qui permet de donner une valeur à une ou à plusieurs variables et qui est de la forme:

$$|| x_1 := x_2 := \ldots := x_s := W ||,$$

où x_i sont des variables ou des identificateurs de procédures, W une expression arithmétique ou logique, le signe || := || étant appelé signe d'affectation; il signifie qu'il faut donner à la quantité à gauche du signe la valeur de la quantité écrite à droite. Pour le cas où on trouve à gauche du signe || := || un identificateur de procédure, voir p. 7.7.8.

Il est établi l'ordre suivant d'exécution d'une instruction d'affectation.

1. On calcule toutes les expressions d'indice qui se trouvent à gauche du signe || := || successivement de la gauche vers la droite.
2. On calcule la valeur de l'expression écrite à droite.
3. On affecte cette valeur à toutes les variables indicées écrites à gauche, dont les valeurs d'indices étaient calculées conformément à 1.

Toutes les variables de la partie gauche doivent être du même type.

Si le type de la partie droite est réel, et celui des variables écrites à gauche est entier, alors on donne aux variables la valeur || *entier* $(A + 0.5)$||, où A est la valeur calculée de l'expression arithmétique.

7.7.4. Instruction de branchement. On appelle *instruction de branchement* une notation de la forme

|| **go to** W||,

où W est une expression de désignation (voir p. 7.6.3). L'exécution de l'instruction de branchement se ramène au calcul de la valeur de l'expression W. Si l'on obtient une étiquette, la commande passe à l'instruction ayant cette étiquette. On la cherche d'abord dans le même bloc que l'instruction de branchement. Si l'on n'y trouve pas l'étiquette en question, on la cherche dans le bloc contenant ce bloc, et ainsi de suite, jusqu'à ce qu'elle soit trouvée.

Si le calcul de W donne un identificateur d'aiguillage et la valeur d'une expression d'indice (un entier sans signe), alors, de même qu'on cherchait l'étiquette dans le cas précédent, on se met à la recherche de la déclaration d'aiguillage d'après son identificateur *) et on agit en conséquence.

7.7.5. Déclaration d'aiguillage. Une déclaration d'aiguillage représente une suite de symboles qui se compose d'un déclarateur ||**switch**||, d'un identificateur d'aiguillage, d'un séparateur ||:=|| et d'une liste d'aiguillage composée d'expressions de désignation séparées par des virgules.

EXEMPLE 7.17. Voici des exemples de déclarations d'aiguillage:

||**switch** $R: = M1, M2, M3$, *entrée* 2||,

switch *passage*$: =$ **if** $x > 1$ **then** *vérification* **else** *sortie*,

if $x < 5$ **then** $M[k + 1]$ **else** $M[k + 2]$||

Après avoir trouvé la déclaration d'aiguillage, on choisit dans la liste d'aiguillage une nouvelle expression de désignation dont le numéro, si l'on parcourt la liste de la gauche vers la droite, vaut la valeur déjà calculée de l'expression d'indice. On calcule ensuite la valeur de la nouvelle expression de désignation et l'on reprend le processus décrit plus haut.

Dans le cas où l'expression de désignation se trouvant dans le bloc extérieur contient des étiquettes, des identificateurs de variables, d'aiguillage ou de procédures déclarés également dans le bloc

*) On ne dit pas dans [1] comment faire si l'étiquette (la déclaration d'aiguillage) n'est pas trouvée. Il faut convenir que l'exécution de l'algorithme s'arrête sans résultat.

interne, on pose (provisoirement) que ces identificateurs ont les valeurs qui leur avaient été affectées en cours d'exécution du bloc extérieur. Définitivement, on obtiendra une étiquette dont la recherche s'effectue de la manière décrite plus haut.

Exemple 7.18. Considérons un morceau de programme qui représente un bloc contenant un autre bloc:

||**begin real** x; **integer** i; **switch** N := **if** $x > 0$ **then**
$M2$ **else** $M3$, N [i]; . . . x := 1; i: = 1;
begin real x; **integer** i; . . . x := —1; i := 2;
go to N [i]; $M2$: x := 3; . . .
end; . . .
end; . . .

Lors de l'exécution de l'instruction de branchement || **go to** N[i]|| se trouvant dans le bloc intérieur, le calcul de l'expression de désignation || N [i]|| fournit || N [2]|| (puisque, immédiatement avant l'exécution de l'instruction de branchement, on a donné à la variable || i|| la valeur 2). La déclaration d'aiguillage avec l'identificateur || N|| se trouve dans le bloc extérieur. Nous devons prendre le deuxième élément dans la liste d'aiguillage || N||. Cet élément représente l'expression de désignation || N [i]||. Bien que la variable || i|| soit déclarée dans le bloc intérieur et y vaille 2, elle valait 1 dans le bloc extérieur. Par conséquent, en calculant la valeur de la dernière expression de désignation nous aurons || N [1]||. Cela veut dire qu'il faut revenir à l'aiguillage || N|| et choisir cette fois dans sa liste la première expression de désignation, à savoir

|| **if** $x > 0$ **then** $M2$ **else** $M3$||.

La variable || x||, qui avait la valeur —1 dans le bloc intérieur, vaut 1 dans le bloc extérieur. Par conséquent, la valeur de la dernière expression de désignation sera l'étiquette || $M2$||. Cherchons-la d'abord dans le bloc intérieur, car c'est là que se trouve l'instruction de branchement. Nous y trouvons cette étiquette. Passons à l'exécution de l'instruction d'affectation || x: = 3;||.

7.7.6. Instruction procédure. Une *instruction procédure* se compose d'un identificateur de procédure et, éventuellement, d'une liste de paramètres effectifs mise entre parenthèses. Les paramètres effectifs sont séparés par des virgules ou par les délimiteurs de paramètres de la forme

||) . . . (:||,

comme dans un indicateur de fonction. Un paramètre effectif peut être soit une chaîne, soit une expression (en particulier, une constante ou une variable), soit un identificateur de tableau, celui d'aiguillage ou de procédure. L'exécution d'une instruction procédure, en grands traits, consiste en ceci. On détermine les valeurs des paramètres effectifs. Parmi les descriptions figurant dans le bloc qui contient l'instruction procédure en question, on cherche la déclaration de procédure correspondante (d'après son identificateur). S'il n'y en a pas, on la cherche dans le bloc qui contient le bloc donné, et ainsi de suite, jusqu'à ce qu'elle soit trouvée. Ceci fait, on exécute la procédure conformément à la déclaration trouvée.

7.7.7. **Déclaration de procédure.** Une *déclaration de procédure* comprend deux parties: la *tête de procédure* et le *corps de procédure*, séparées par le symbole ||;||. La tête de procédure comporte le symbole de base || **procedure** || ou bien ce symbole précédé d'une indication de type, donc: || **integer procedure** || **real procedure** || **Boolean procedure** || et l'identificateur de procédure; il peut être suivi d'une liste de paramètres formels mise entre parenthèses, d'une liste de valeurs (partie VALEUR) et d'une ou plusieurs spécifications (partie SPECIFICATION).

Une *liste de paramètres formels* représente une suite d'identificateurs séparés par des délimiteurs de paramètres, qui sont soit des virgules, soit des notations de la forme ||) . . . : (||, où les points désignent des lettres quelconques.

Une *liste de valeurs* est une suite se composant d'un descripteur || **value** || et d'une liste d'identificateurs séparés par des virgules. Seuls les identificateurs qui figurent dans la liste de paramètres formels peuvent être placés dans la liste de valeurs.

Une *spécification* commence par l'un des spécificateurs:

|| **string** || **label** || **switch** || **array** || **procedure** ||
integer || **real** || **Boolean** || **integer array** ||
real array || **Boolean array** || **integer procedure**
|| **real procedure** || **Boolean procedure** ||

Le reste de la spécification représente une liste d'identificateurs (séparés par des virgules) qu'on trouve dans la liste de paramètres effectifs.

La liste de paramètres formels, celle de valeurs et les spécifications sont séparées par des séparateurs ||;||.

La tête de procédure peut ne pas contenir de paramètres formels. Or, dans la liste de valeurs peuvent figurer seuls les identificateurs qui sont des paramètres formels de la procédure. D'autre part, une spécification n'est obligatoire que pour les paramètres formels qui figurent dans la partie VALEUR, donc une tête de procédure qui

ne contient pas de liste de paramètres formels ne peut pas contenir de listes de valeurs ni spécifications. Un même paramètre formel ne doit pas figurer dans deux spécifications à la fois.

Le *corps de procédure* représente une instruction quelconque.

L'*exécution d'une procédure* consiste en ce qui suit. On attribue aux paramètres formels de la procédure cités dans la liste de valeurs les valeurs des paramètres effectifs obtenues en exécutant l'instruction procédure. Les autres paramètres formels de la procédure sont remplacés dans le corps de procédure par les paramètres effectifs respectifs. La correspondance entre les paramètres formels de la procédure et les paramètres effectifs indiqués dans l'instruction procédure s'obtient en faisant correspondre un à un, dans le même ordre, les éléments de la liste de paramètres effectifs et ceux de la liste de paramètres formels.

Les quantités pouvant figurer dans le corps de procédure sont de quatre sortes:

a) celles qui sont décrites dans les blocs contenus dans le corps de procédure; elles sont dites locales pour la procédure donnée;

b) celles qui sont paramètres effectifs contenus dans la liste de valeurs; elles sont dites « variables *value* »;

c) celles qui sont paramètres effectifs non indiqués dans la liste de valeurs; elles sont dites « variables name »;

d) quantités non locales déclarées à l'extérieur de l'instruction de procédure donnée, dans un bloc plus grand ou même en tête du bloc où se trouve la déclaration de procédure.

En cours d'exécution d'une procédure, il peut arriver qu'on aura à substituer à une variable *name* un identificateur (obtenu en calculant un paramètre effectif) identique à l'identificateur désignant une variable locale. Dans ce cas il faut choisir un identificateur quelconque ne se rencontrant pas dans le programme ALGOL et remplacer, dans les déclarations et dans les instructions du bloc où ceci s'est produit, l'identificateur de la variable locale par ce nouvel identificateur. Ensuite on peut remplacer la variable name par le paramètre formel qui lui correspond.

Exemple 7.19. Soit une déclaration de procédure:

|| **procedure** F (x, y), **value** x; **begin real** z;

$z := x \uparrow 2$; $y := z + 1$ **end**||.

La procédure décrite permet, d'après un x donné, de calculer la quantité $x^2 + 1$ et de l'attribuer à celui des paramètres effectifs qui correspondra au paramètre formel || y||.

La procédure est ici || F (x, z)||. Si l'on ne tenait pas compte du fait que le paramètre effectif || z|| coïncide avec la variable || z||

localisée dans le corps de la procédure, on obtiendrait l'instruction

|| **begin** **real** z; z: $= x \uparrow 2$; z: $= z + 1$ **end**||

qui n'aurait pas de résultat, car la quantité locale || z|| est perdue en sortant du bloc où elle était décrite (voir p. 7.7.2 et § 7.9). Par contre, si l'on remplace préalablement, dans le corps de la procédure, || z|| par || u|| par exemple, on obtient d'abord

|| **begin** **real** u; u: $= x \uparrow 2$; y: $= u + 1$ **end**||

et puis

|| **begin** **real** u; u: $= x \uparrow 2$; z: $= u + 1$ **end**||.

La dernière instruction, comme il le faut, calcule d'après la valeur de x celle de $x^2 + 1$ et affecte cette dernière à la variable || z ||.

Les opérations figurant dans le corps de procédure modifié comme décrit s'exécutent, ce qui achève l'exécution de la procédure. On considère que les variables non locales du corps de procédure ont les valeurs qui leur étaient propres dans le bloc contenant la déclaration de la procédure, et que les paramètres effectifs correspondant aux paramètres formels *name* ont les valeurs qui leur étaient propres dans le bloc contenant l'instruction de la procédure.

Après l'exécution de la procédure on retourne à l'instruction qui suit immédiatement l'instruction procédure ayant déclenché la procédure en question, si seulement son effet n'est pas autre.

7.7.8. Calcul de valeur d'une fonction. La déclaration d'une procédure destinée à calculer la valeur d'une fonction commence par l'un des spécificateurs suivants:

|| **integer procedure** || **real procedure** || **Boolean procedure** ||.

Le corps de procédure doit contenir une instruction d'affectation qui donne à l'identificateur de cette procédure la valeur égale à la valeur calculée de la fonction. Une telle instruction peut exister seulement dans le corps de la procédure donnée. Le type de la valeur qu'on attribue à l'identificateur de procédure doit correspondre au premier spécificateur qu'on rencontre en tête de la procédure.

7.7.9. Procédures-codes. Procédures d'entrée-sortie. Le langage ALGOL admet des procédures exprimées en un langage autre que l'ALGOL. On suppose que quiconque utilisant de telles procédures comprend le langage employé. Outre les expressions et les identificateurs, c'est encore les chaînes qui peuvent être utilisées comme paramètres effectifs de telles procédures.

La description d'une telle procédure comprend l'en-tête construit d'après les règles générales et le corps de procédure auquel on

n'impose aucune restriction dans le langage ALGOL. Comme certains paramètres effectifs de la procédure décrite peuvent être des chaînes, la partie spécification contiendra alors des spécifications d'une forme particulière, notamment, un symbole || **string** || suivi d'une liste d'identificateurs séparés par des virgules. Aux paramètres formels de cette liste il doit correspondre dans l'instruction procédure les paramètres effectifs qui sont des chaînes.

Les chaînes peuvent être également des paramètres effectifs d'une procédure décrite en ALGOL, si le corps de cette procédure contient l'appel d'une procédure décrite en un langage autre que l'ALGOL.

EXEMPLE 7.20. Soit une déclaration de procédure

|| **procedure** *U*1 (*a*, *y*) ; **value** *a* ; **real** *a* ; **begin** . . . ; *U*2 (*y*) ; . . . **end** ||.

Supposons que le paramètre formel *y* ne se rencontre dans le corps de la procédure *U*1 qu'en tant que paramètre effectif de la procédure *U*2 qui n'est pas décrite en ALGOL. Alors le paramètre effectif de l'instruction procédure *U*1 qui correspond au paramètre formel *y* peut être une chaîne. Par exemple, l'instruction procédure mentionnée peut avoir la forme

|| *U*1 (0.512, '0025 ␣ 0031 ␣ 0421 ␣ 0 ␣ 01') ||.

La chaîne qui en fait partie représente une instruction de machine.

Comme on a vu au p. 7.6.3, une variable ne peut être utilisée pour le calcul d'une expression que si elle a obtenu, vers le moment de son utilisation, une valeur déterminée. Les variables dont les valeurs ne sont pas calculées, mais sont utilisées dans les calculs, doivent recevoir leurs valeurs à partir des données initiales, gardées dans une mémoire extérieure. A cette fin sont prévues dans l'ALGOL les procédures d'entrée qui sont des procédures-codes:

|| **procedure** *Entrée du symbole* (*x*) *dans l'alphabet*: (*y*) *du canal*: (*n*) ; **value** *n* ; **integer** *x*, *n* ; **string** *y* ; ⟨corps de procédure⟩ ||.

|| **procedure** *Sortie du symbole* (*x*) *dans l'alphabet*: (*y*) *dans le canal*: (*n*) ; **value** *x*, *n* ; **integer** *x*, *n* ; **string** *y* ; ⟨corps de procédure⟩ ||.

|| **procedure** *Entrée de l'entier* (*x*) *du canal*: (*n*) ; **value** *n* ; **integer** *x*, *n* ; ⟨corps de procédure⟩ ||.

|| **procedure** *Sortie de l'entier* (*x*) *dans le canal*: (*n*) ; **value** *x*, *n* ; **integer** *x*, *n* ; ⟨corps de procédure⟩ ||.

|| **procedure** *Entrée du nombre* (*x*) *du canal*: (*n*) ; **value** *n* ; **real** *x* ; **integer** *n* ; ⟨corps de procédur›⟩ ||.

|| **procedure** *Sortie du nombre* (*x*) *dans le canal*: (*n*) ; **value** *n* ; **real** *x* ; **integer** *n* ; ⟨corps de procédure⟩ ||.

|| **procedure** *Entrée du tableau* (x) *du canal*: (n); **value** n; **integer** n; **array** x; ⟨corps de procédure⟩||.

|| **procedure** *Sortie du tableau* (x) *dans le canal*: (n); **value** n; **integer** n; **array** x; ⟨corps de procédure⟩||.

Soulignons que le corps de chacune des procédures standard citées ne doit pas contenir de variables non locales, ni d'instructions de branchement qui mènent à l'extérieur (la signification de cette condition devient claire si l'on tient compte du p. 7.7.10 et du § 7.10). Chacune de ces procédures convertit la représentation interne dans la représentation externe (ou le contraire) de la manière suivante. Une procédure d'entrée attribue les valeurs aux éléments de l'état de mémoire intérieure, en effaçant dans l'ordre le contenu du canal dont le numéro est donné dans l'appel de la procédure d'entrée. Si, lors de l'opération d'entrée, un canal est épuisé, il est complètement éliminé de la mémoire extérieure. Une procédure de sortie, sans modifier l'état de mémoire intérieure, ajoute les notations correspondantes (symboles, entiers, nombres, suites de symboles, d'entiers ou de nombres) à la fin du contenu du canal dont le numéro est donné dans l'appel de la procédure. Si l'état de mémoire extérieure ne contient pas le canal de numéro qui correspond au paramètre formel || n || de la procédure de sortie, alors un tel canal est organisé, et l'information de sortie devient son contenu.

Une procédure d'entrée qui ne trouve pas le canal demandé s'arrête sans résultat.

Arrêtons-nous spécialement sur les procédures d'entrée-sortie de symboles. Une procédure d'entrée, après avoir effacé le premier symbole du canal correspondant, trouve dans la chaîne y donnée dans l'appel de la procédure le symbole identique à celui effacé. Ensuite elle donne à la variable x le numéro égal à celui du symbole trouvé dans la chaîne y. Ainsi, le résultat de l'entrée du symbole représente une valeur d'une variable simple du type **integer**. Une procédure de sortie de symboles perçoit la valeur de la variable simple du type **integer** qui correspond au paramètre formel x comme le numéro du symbole (en comptant de la gauche vers la droite) dans la chaîne correspondant au paramètre formel y. La procédure de sortie de symboles ajoute le symbole indiqué à la fin du contenu du canal correspondant (à sa droite).

Ainsi, parmi les valeurs des éléments de l'état de mémoire intérieure, les symboles figurent en tant que numéros des lettres d'un alphabet donné sous forme d'une ligne. Ils peuvent être traités au moyen de tout l'arsenal des opérations logiques et mathématiques de l'ALGOL, mais, pour être sortis, doivent être mis de nouveau sous forme de numéros des symboles d'un alphabet (disposé en ligne).

On n'a pas besoin d'inclure les descriptions des procédures standard, en particulier de celles d'entrée et de sortie, dans des blocs.

On convient qu'elles sont toujours à la portée d'un programme ALGOL (voir § 7.9).

Exemple 7.21. Le bloc suivant d'un programme ALGOL sert d'illustration d'utilisation des procédures d'entrée-sortie:

|| **begin real** $a1$; **integer** $a2$, $b1$, $b2$, $b3$; **real array** c [5 : 50] ; *Entrée nombre* ($a1$, $b1$) ; *Entrée entier* ($a2$, $b2$) ; *Entrée tableau* (c, $b3$) ; c [$a2$] : = = c [$a2$] + $a1$; *Sortie tableau* (c, $b3$) **end** ||.

Pour mieux comprendre les actions déterminées par ce bloc, il est bon d'exprimer le texte ALGOL dans un français naturel: « Faire entrer le nombre $a1$ du canal n° $b1$, l'entier $a2$ du canal n° $b2$, le tableau de nombres c [5 : 50] du canal n° $b3$. Augmenter de $a1$ la valeur de l'élément c [$a2$] du tableau c, après quoi délivrer le tableau obtenu (qui a le même nom c [5 : 50]) dans le canal n° $b3$ ».

Au cours de l'exécution du bloc décrit, les 46 premiers nombres du contenu du canal n° $b3$ seront d'abord effacés, puis écrits (après la modification de l'un d'eux) à la fin du même canal n° $b3$.

Exemple 7.22. L'élément de mémoire extérieure

$$\| \textit{Canal } 25 \equiv -13.220\nabla 10.25_0 \|,$$

après l'exécution de la procédure

$$\| \textit{Entrée nombre } (x,\ 25) \|$$

devient

$$\| \textit{Canal } 25 \equiv 10.25_0 \|.$$

De plus, un élément de l'état de mémoire intérieure prendra la valeur 13.220. Si l'on veut exécuter ensuite la procédure

$$\| \textit{Sortie nombre } (x,\ 26) \|$$

et si le canal numéro 26 est inexistant dans la mémoire extérieure, il se crée l'élément

$$\| \textit{Canal } 26 \equiv -13.220_0 \|$$

de l'état de mémoire extérieure.

7.7.10. Effet de bord. Lors de l'exécution d'une procédure, il peut se produire une modification imprévue de la valeur d'une variable qui n'y est pas localisée. Ce phénomène s'appelle effet de bord.

L'effet de bord n'était pas spécialement prévu par les auteurs de l'ALGOL. Cependant les programmeurs en tirent certains avantages. Tout de même, il faut convenir que les corps de procédures doivent être composés de façon à éviter cet effet.

EXEMPLE 7.23. Considérons le bloc

|| **begin real** x, y, u ; **real procedure** f (u) ; **value** u ; **begin** x := $u \uparrow 2$;
f : = $x + 1$ **end**; . . . y: = $x + f(x) + x$; . . . **end** ||.

Dans l'instruction

$$|| \; y: = x + f(x) + x \; ||$$

l'effet de bord qui se manifeste lors du calcul de f (x) et qui consiste en une modification du paramètre effectif x amènerait à ce que la valeur du dernier terme x différerait de celle du premier terme x.

Il faut en tenir compte en programmant en ALGOL, ainsi qu'en élaborant les compilateurs de ce langage en des langages machine.

L'effet de bord ébranle en quelque sorte l'harmonie de l'ALGOL. Il peut arriver qu'au cours de l'exécution d'opérations intervenant dans les expressions contenues dans les instructions de base, il se produise des modifications de valeurs de variables. Or, selon l'idée du langage, c'est une prérogative des instructions (et non des opérations).

7.7.11. **Instruction vide.** Une instruction vide correspond à l'absence d'instruction. On ne l'écrit pas. Elle présente deux particularités : a) son exécution se ramène au passage à l'instruction suivante ; b) on peut l'étiqueter.∎

7.7.12. **Instruction de boucle (instruction FOR).** Une instruction de boucle se compose de l'*en-tête* et d'une instruction quelconque à exécuter plusieurs fois (en particulier, une fois ou pas une seule fois).

L'en-tête d'une boucle représente une suite se composant de l'opérateur séquentiel || **for** ||, d'une variable qu'on appelle paramètre de la boucle ou variable contrôlée, d'un signe d'affectation, d'une liste de boucle et de l'opérateur séquentiel || **do** ||.

On appelle *liste de* FOR une suite d'éléments de la liste de FOR séparés par des virgules. Il y a trois types d'éléments d'une liste de FOR. Un élément du premier type est une expression arithmétique (en particulier, une variable ou un nombre). Un élément du deuxième type est une «*progression arithmétique*» ; il se compose de trois expressions arithmétiques séparées par des symboles de base || **step** || et || **until** ||. Cet élément a la forme

|| $A1$ **step** $A2$ **until** $A3$ ||,

où $A1$, $A2$ et $A3$ sont des expressions arithmétiques quelconques. Un élément du troisième type, dit « *itératif* », se compose d'une expression arithmétique, d'un séparateur || **while** || et d'une expression logique. Il a la forme

|| A **while** P ||,

où A et P sont respectivement une expression arithmétique et une expression logique. Une liste de FOR peut contenir juxtaposés des éléments des trois types étudiés.

L'exécution d'une instruction FOR consiste en ce qui suit. On affecte une valeur à la variable de contrôle, ensuite on exécute l'instruction intérieure de boucle, après quoi, si c'est possible, on affecte une nouvelle valeur à la variable de contrôle, et ainsi de suite, jusqu'à ce que l'affectation de valeur à la variable de contrôle devienne impossible. La boucle étant exécutée, le contrôle passe à l'instruction immédiatement suivante.

L'affectation de valeurs à la variable de contrôle s'effectue au moyen des éléments de la liste de FOR. On commence par le premier élément. Les moyens de ce premier élément épuisés, on passe à l'élément suivant. Lorsque le dernier élément sera traité à bout, la variable de contrôle ne pourra prendre aucune nouvelle valeur jusqu'à un nouvel appel de l'instruction FOR considérée.

Dans le cas d'un élément du premier type on a à calculer une fois l'expression arithmétique et à attribuer sa valeur à la variable contrôlée.

S'il s'agit d'un élément du deuxième type, on devra calculer les valeurs de trois expressions arithmétiques, dont la première est considérée comme le premier terme, la deuxième, comme la raison et la troisième, comme le dernier terme d'une progression arithmétique, et affecter à la variable de contrôle les valeurs de tous les termes de la progression arithmétique successivement.

Dans le cas d'un élément du troisième type on calcule la valeur de l'expression arithmétique qu'on affecte à la variable de contrôle; ensuite on calcule la valeur de l'expression logique et, si la valeur de cette dernière est le **vrai**, l'instruction intérieure de la boucle s'exécute. Si la valeur de l'expression logique est le **faux**, on considère l'effet de l'élément comme terminé. Après l'exécution de l'instruction intérieure le processus est itéré.

Aucune instruction interne étiquetée d'une boucle n'est accessible de l'extérieur; en revanche, il est toujours possible de faire un saut vers n'importe quelle instruction étiquetée à l'extérieur de la boucle.

EXEMPLE 7.24. On peut calculer la valeur d'une somme algébrique

$$y = \sum_{i=1}^{n} a_i$$

au moyen de deux instructions: une instruction d'affectation

$$\| y := 0 \|,$$

et l'une des instructions FOR qui suivent:

$\|$ **for** $i := 1$ **step** 1 **until** n **do** $y := y + a[i] \|$, $\|$ **for** $i := 1$, **while** $j < n + 1$ **do begin** $j := i;\ y := y + a[i];\ j := i + 1$ **end** $\|$.

Dans la première des instructions FOR, la liste de FOR se réduit à un seul élément du deuxième type (« progression arithmétique »). Dans la seconde instruction FOR, la liste de FOR se compose de deux éléments: un élément du premier type et un élément du troisième type (« itératif »).

7.7.13. Instruction IF. C'est une instruction de la forme

|| **if** P **then** Q ||,

où P est une expression logique, Q une instruction inconditionnelle ou une boucle. Son exécution consiste à calculer l'expression logique P et à prendre une décision selon le résultat. Si l'on obtient || **true** ||, on exécute l'instruction Q pour passer ensuite (si cette instruction n'a pas modifié l'ordre d'exécution d'instructions) à l'instruction qui suit immédiatement l'instruction IF traitée. Et si l'on obtient || **false** ||, alors on passe directement à l'exécution de l'instruction qui suit, dans le programme ALGOL, l'instruction IF considérée.

Exemple 7.25. Voici quelques exemples d'instructions IF:

|| **if** $x > y$ **then** $x := x - y$ ||,

|| **if** $x > y \wedge y > z$ **then begin** $x := x - y$; $y := y - z$;

$u := x \times y$ **end** ||, **if** $x > y$ **then begin if** $y > z$ **then** $y := z - y$ **end** ||,

|| **if** $x > y$ **then for** $i := 1$ **step**˽2 **until** 25 **do** $y := y + a\,[i]$ ||.

Dans la première instruction IF l'instruction inconditionnelle est une instruction d'affectation, dans la deuxième et la troisième une instruction composée, et dans la quatrième on a une boucle.

7.7.14. Instruction alternative. On appelle ainsi une instruction de la forme:

|| **if** P **then** R **else** S ||,

où P est une expression logique, R une instruction inconditionnelle (mais non une boucle! *)), S une instruction quelconque.

---

*) Si l'on choisissait pour R une boucle, ceci conduirait à une ambiguïté dans le cas d'une boucle de la forme || Z **if** P' **then** Q ||, où Z est l'en-tête de boucle. On aurait eu la notation || **if** P **then** Z **if** P' **then** Q **else** S || que l'on pourrait interpréter comme

||**if** P **then** $\{Z$ **if** P' **then** Q **else** $S\}$||,

i.e. comme une instruction IF, ou bien comme

|| **if** P **then** $\{Z$ **if** P' **then** $Q\}$ **else** S ||,

i.e. comme une instruction alternative.

Ici, pour montrer les groupements possibles de symboles, nous nous sommes servis des accolades qui ne sont pas des symboles de l'ALGOL.

L'exécution d'une instruction alternative consiste en ce qu'on calcule la valeur de l'expression logique P et, en cas de || true ||, on exécute R, et en cas de || false ||, on exécute S. Si l'exécution de l'instruction considérée n'a pas modifié l'ordre d'exécution d'instructions, on passe à l'instruction qui suit immédiatement l'instruction alternative donnée.

Notons qu'on réunit sous le nom plus général d'*instructions conditionnelles* les instructions IF et les instructions que nous avons appelées ici, par souci de brièveté, alternatives.

§ 7.8. Programme ALGOL. Commentaires

Du point de vue de sa structure, un algorithme donné en ALGOL (un programme ALGOL) représente une instruction composée ou un bloc qui n'est partie d'aucun autre bloc. L'exécution d'un programme ALGOL consiste en l'exécution de cette instruction composée ou de ce bloc. Un programme ALGOL ne peut pas ne pas contenir de blocs, sinon il serait inefficace. En effet, pour un programme il n'existe pas de variables non locales, et il n'existerait pas de variables locales s'il n'y avait pas de blocs, par suite de l'inexistence de descriptions de type et de tableaux (voir § 7.3) qui n'apparaissent que dans les blocs. Même plus, un programme sans blocs n'aurait pas de descriptions de procédure, sans quoi il serait impossible de transférer une valeur de variable de la mémoire extérieure dans la mémoire intérieure de la machine.

On a convenu qu'un programme ALGOL ne comporte pas de descriptions de procédures-fonctions standard (voir § 7.5) ni de procédures d'entrée-sortie (voir p. 7.7.9). Dans ce cas on fait précéder un programme ALGOL des descriptions de procédure qui manquent, et l'on met le tout entre crochets d'instruction. Ces procédures sont considérées comme implicitement présentes dans tout programme ALGOL.

Outre les instructions et les déclarations, on peut inclure dans un programme ALGOL des *commentaires*, qui n'ont aucun rôle dans le déroulement du programme mais qui facilitent la lecture. Il existe quatre modes d'introduire un commentaire dans un programme:

1. un texte arbitraire, qui commence par le symbole || comment ||, se termine par un point-virgule et qui ne contient pas à l'intérieur de points-virgules, peut être mis après n'importe quel symbole || begin || ou après n'importe quel point-virgule;

2. un texte arbitraire qui ne comporte pas de symbole || end || ni point-virgule peut être inséré entre n'importe quel symbole || end || et le symbole suivant;

3. un texte arbitraire qui se compose de lettres et de chiffres mais commence par une lettre (i.e. tout comme un identificateur)

peut être placé comme étiquette devant une instruction (étiquetée ou non);

4. un texte composé uniquement de lettres peut être inclus dans la liste de paramètres formels d'une procédure (voir p. 7.7.7.) ou dans la liste de paramètres effectifs d'une instruction procédure ou d'un indicateur de fonction (p.p. 7.7.6, 7.7.8) en tant que délimiteur de paramètre.

§ 7.9. Algorithme d'exécution d'un programme ALGOL

Comme il est difficile de se faire une idée claire du processus d'exécution d'un programme ALGOL d'après seules les descriptions des instructions, nous allons décrire en détail l'algorithme d'exécution d'un programme ALGOL. Nous convenons que tous les éléments du programme ALGOL y soient présents explicitement (voir § 7.8). De plus, nous supposons que toutes les données initiales du programme soient gardées dans la mémoire extérieure (§ 7.4).

Dans la description de l'algorithme nous nous servirons des termes suivants : noyau de procédure, noyau de boucle, noyau d'instruction IF, noyau d'instruction alternative. La signification du terme *noyau de procédure* est expliquée au p. 30° de l'algorithme d'exécution. Nous appelons *noyau d'une boucle* l'instruction qui suit l'en-tête de la boucle. Nous appelons *noyau d'une instruction* IF et *noyau d'une instruction alternative* l'instruction qui y figure après l'opérateur séquentiel || **then** ||. On trouvera quelques éclaircissements au § 7.10.

ALGORITHME D'EXÉCUTION D'UN PROGRAMME ALGOL

1°. Examiner le programme. Numéroter tous les blocs où apparaissent les déclarateurs || **own** || (nous supposons que les corps de procédures n'en contiennent pas). Les numéros ainsi obtenus seront considérés comme les numéros propres des blocs correspondants. Considérer tout le programme ALGOL comme une instruction. Passer au p. 2°.

2°. Vérifier si l'instruction donnée est composée. Si oui, aller en 22°, si non, passer au p. 3°.

3°. Voir si l'instruction donnée est un bloc. Si oui, aller en 23°, si non, passer au p. 4°.

4°. Voir si l'instruction examinée est une instruction d'affectation. Si oui, aller en 24°, si non, passer au p. 5°.

5°. Voir si l'instruction donnée est une instruction de branchement. Si oui, aller en 25°, si non, passer au p. 6°.

6°. Voir si l'instruction donnée est une instruction procédure. Si oui, aller en 29°, si non, passer au p. 7°.

7°. Voir si l'instruction donnée est vide. Si oui, aller en 10°, si non, passer au p. 8°.

8°. Voir si l'instruction donnée est une instruction FOR. Si oui, aller en 36°, si non, passer au p. 9°.

9°. Voir si l'instruction donnée est une instruction IF. Si oui, aller en 20°, si non, aller en 42°.

10°. Voir si l'instruction donnée est la dernière dans le noyau de procédure (qui n'est pas une fonction). Si oui, aller en 34°, si non, passer au p. 11°.

11°. Voir si l'instruction donnée est la dernière dans le noyau de boucle. Si oui, aller en 36°, si non, passer au p. 12°.

12°. Voir si l'instruction donnée est la dernière dans le noyau d'une instruction alternative. Si oui, aller en 19°, si non, passer au p. 13°.

13°. Voir si l'instruction donnée est la dernière dans le noyau d'une procédure-fonction. Si oui, aller en 32°, si non, passer au p. 14°.

14°. Voir si l'instruction donnée est la dernière instruction d'un bloc. Si oui, passer au p. 15°, si non, aller en 18°.

15°. On suppose que le bloc considéré est celui où l'instruction donnée était la dernière. Effacer tous les éléments de mémoire intérieure dont les niveaux correspondent au niveau qui avait lieu à l'intérieur de ce bloc, exception faite des éléments qui correspondent aux variables décrites au moyen du déclarateur || **own** ||. Passer au p. 16°.

16°. Vérifier si l'instruction donnée est la dernière instruction du programme. Si oui, passer au p. 17°, si non, aller en 21°.

17°. Fin de l'analyse du programme. Effacer tous les éléments d'état de mémoire intérieure qui y subsistent. Considérer comme résultat d'exécution de l'algorithme l'état de mémoire extérieure.

18°. Poser que l'on considère l'instruction composée dans laquelle l'instruction donnée était la dernière. Aller en 16°.

19°. Examiner dans son ensemble l'instruction alternative dont le noyau était l'instruction donnée. Revenir en 16°.

20°. Calculer la valeur de l'expression logique contenue dans l'instruction IF. Au cours de ce calcul, on pourra avoir besoin de la valeur d'une fonction. Pour obtenir cette valeur, interrompre le calcul de l'expression et aller en 35°. Après avoir obtenu la valeur de l'expression logique, aller en 31°.

21°. Passer à l'examen de l'instruction suivante. Revenir en 2°.

22°. Considérer la première instruction interne de l'instruction composée donnée. Revenir en 2°.

23°. Pour chaque variable déclarée dans le bloc sans utilisation de déclarateur || **own** ||, prévoir un élément de mémoire intérieure pour une variable simple, et un tableau d'éléments de même nom que la variable s'il s'agit d'une variable indicée, le niveau des éléments dépassant d'une unité le plus grand de deux nombres, dont l'un est la quantité de niveaux propres, et l'autre, le niveau maximal des éléments de même nom qui se trouvent déjà en mémoire intérieure.

Poser les valeurs de ces éléments vides. Pour chaque variable décrite dans le bloc à l'aide d'un déclarateur || **own** ||, s'il n'y a pas de mémoire intérieure qui contienne déjà les éléments correspondants, il y a création d'une nouvelle mémoire de manœuvre de la manière décrite plus haut, avec la seule différence que le niveau qu'on attribue à cette mémoire est égal au niveau propre du bloc. Considérer la première instruction interne du bloc. Aller en 2°.

24°. Exécuter l'instruction d'affectation de la façon décrite en 7.7.3. Supposer en le faisant que les éléments qui correspondent dans la mémoire intérieure aux variables utilisées aient les niveaux maximaux ou égaux au niveau propre du bloc qui contient l'instruction d'affectation. En rencontrant au cours du calcul de l'expression contenue dans l'instruction d'affectation un indicateur de fonction dont les paramètres effectifs ont déjà des valeurs déterminées, interrompre le calcul de l'expression et aller en 35°. Une fois l'instruction d'affectation exécutée, passer au p. 10°.

25°. Calculer l'expression de désignation apparaissant dans l'instruction de branchement donnée en supposant que tous les éléments de l'état de mémoire intérieure qui correspondent aux variables utilisées aient les niveaux maximaux ou égaux au niveau propre du bloc contenant cette instruction de branchement. En rencontrant au cours du calcul de l'expression un indicateur de fonction dont tous les paramètres effectifs ont déjà des valeurs déterminées, interrompre le calcul de l'expression et aller en 35°. Après le calcul de l'expression de désignation passer au p. 26°.

26°. Voir si la valeur obtenue de l'expression de désignation est une étiquette. Si oui, passer au p. 27°, si non, au p. 28°.

27°. Par la méthode décrite au p. 7.7.4, trouver dans le programme ALGOL l'instruction étiquetée avec l'étiquette donnée. Chaque fois que l'on quitte un bloc sans y trouver l'étiquette cherchée, on effacera les éléments de l'état de mémoire intérieure qui correspondent aux variables déclarées dans ce bloc, à l'exception de variables propres (décrites à l'aide d'un déclarateur || **own** ||). Se mettre à l'examen de l'instruction trouvée. Revenir en 2°.

28°. Trouver (d'après l'identificateur d'aiguillage obtenu) la déclaration d'aiguillage par la méthode décrite au p. 7.7.4. En le faisant, compter le nombre k de sorties du bloc contenant l'instruction de branchement dans les blocs qui l'englobent.

Choisir dans la déclaration d'aiguillage trouvée l'élément de la liste d'aiguillage dont le numéro d'ordre (en comptant de la gauche vers la droite) est égal à la valeur de l'expression en indice *). Cal-

*) Selon [1], lorsqu'il n'y a pas d'élément cherché dans la liste d'aiguillage, on pose que l'on considère l'instruction de branchement qui vient d'être exécutée et on passe au p. 10°. Cette règle ne nous paraît pas logique. A notre avis, l'absence d'élément nécessaire dans la liste d'aiguillage devait conduire à l'interruption sans résultat du processus algorithmique.

culer l'expression de désignation contenue dans l'élément choisi de la liste d'aiguillage pour chaque variable non locale et les éléments de mémoire intérieure dont les niveaux sont diminués d'un nombre entier k par rapport aux niveaux maximaux des éléments de même nom, ou sont égaux au niveau propre du bloc où apparaît la déclaration d'aiguillage.

Lorsqu'en calculant l'expression de désignation on rencontre un indicateur de fonction dont les paramètres effectifs sont déjà calculés, il faut interrompre le calcul de l'expression de désignation et aller au p. 35°. La valeur obtenue de l'expression de désignation sera considérée comme la valeur de l'expression de désignation de l'instruction de branchement en train d'exécution et revenir au p. 26°.

29°. Exécuter l'instruction procédure de la manière décrite en 7.7.6. On obtiendra d'abord les valeurs des paramètres effectifs, puis on trouvera la déclaration de procédure. Au cours du calcul des paramètres effectifs, le calcul d'un indicateur de fonction peut s'avérer nécessaire. Interrompre dans ce cas les calculs et passer au p. 35°. Après le calcul des paramètres effectifs chercher la déclaration de procédure. Au cours de cette recherche il faut déterminer les niveaux des éléments de l'état de mémoire intérieure qui correspondent au bloc contenant la déclaration en question. Passer au p. 30°.

30°. D'après la déclaration de procédure trouvée, construire l'instruction de la manière suivante. Remplacer dans le corps de procédure les paramètres formels qui ne sont pas indiqués dans la liste de valeurs dans la tête de procédure par les paramètres effectifs correspondants. Construire le noyau de procédure de la manière suivante. Ecrire après le symbole || **begin** || les déclarations des types des variables et des tableaux qui représentent les paramètres formels figurant dans la liste des valeurs ; en le faisant utiliser des spécifications ; écrire ensuite les instructions d'affectation qui donnent aux paramètres formels de la liste des valeurs les valeurs des paramètres effectifs ; écrire ensuite le corps de procédure modifié comme décrit plus haut que l'on fera suivre du symbole || **end** ||. Considérer, pendant son exécution, le noyau de procédure obtenu comme écrit à la place de l'instruction qui a appelé la procédure. On suppose que les variables qui étaient non locales dans le corps de procédure correspondent aux éléments de l'état de mémoire intérieure qui sont liés au bloc contenant la déclaration de procédure. Aller en 2°.

31°. Vérifier si la valeur obtenue est un || **true** ||. Si oui, aller en 40°, si non, aller en 41°.

32°. Poser ξ (voir le dernier alinéa du § 7.10) égal au dernier numéro de point apparaissant dans la chaîne A. Barrer dans A ce numéro. Passer au p. 33°.

33°. Effacer le noyau de procédure exécuté. Supprimer les éléments de l'état de mémoire intérieure correspondant aux variables et aux tableaux qui étaient localisés dans le noyau de la procédure-

fonction (il n'y a pas de propres). Poser que l'indicateur de fonction qui a provoqué son calcul a la valeur affectée à l'identificateur de la procédure-fonction. Revenir au calcul de l'expression appartenant à l'instruction d'affectation, de procédure, de branchement, d'aiguillage, FOR, IF ou alternative qui a été interrompue pour calculer la valeur de la fonction. (Passer respectivement au p.p. 24°, 29°, 25°, 28°, 36°, 20° ou 42°, c'est-à-dire revenir au p. ξ°.)

34°. Effacer le noyau exécuté de la procédure non fonction. Effacer les éléments de l'état de mémoire intérieure correspondant aux variables et aux tableaux qui étaient localisés dans le noyau de procédure. Considérer l'instruction procédure qui a appelé la procédure. Revenir au p. 21°.

35°. Chercher la déclaration de la procédure-fonction à l'intérieur du bloc contenant l'instruction qui a nécessité le calcul de la fonction. Si la description n'est pas trouvée dans ce bloc, passer au bloc qui l'englobe, etc. Après avoir trouvé la déclaration cherchée, inclure dans la chaîne A le numéro du point à partir duquel est venu l'appel du point donné, revenir au p. 30°.

36°. Considérer la tête de boucle. Si le calcul des expressions qui y sont contenues nécessite de déterminer la valeur d'une fonction, exécuter le p. 35°. Après le calcul de toutes les expressions nécessaires passer au p. 37°.

37°. Déterminer comme indiqué en 7.7.12 (en passant éventuellement par le p. 35°) si l'instruction (le noyau) de boucle doit être exécutée. Si oui, passer au p. 38°, si non, passer au p. 39°.

38°. Poser que l'on considère le noyau de boucle. Revenir au p. 2°.

39°. Considérer la boucle dans son ensemble. Revenir au p. 10°.

40°. Considérer le noyau de l'instruction IF. Revenir au p. 2°.

41°. Considérer l'instruction IF dans son ensemble, exécuter le p. 10°.

42°. Calculer la valeur de l'expression logique contenue dans l'instruction alternative. On pourra avoir besoin de la valeur d'une fonction. Pour l'obtenir, interrompre le calcul de l'expression et exécuter p. 35°. Après avoir calculé la valeur de l'expression logique, passer au p. 43°.

43°. Vérifier si la valeur obtenue est un || **true** ||. Si oui, passer au p. 44°, si non, aller en 45°.

44°. Considérer l'instruction qui suit, dans l'instruction alternative, l'opérateur séquentiel || **then** ||. Revenir au p. 2°.

45°. Considérer l'instruction qui suit, dans l'instruction alternative, l'opérateur séquentiel || **else** ||. Revenir au p. 2°.

En examinant l'algorithme d'exécution d'un programme ALGOL nous voyons que la donnée initiale pour ce programme est un état de mémoire extérieure, i.e. plusieurs canaux qui contiennent des suites de symboles et de nombres. En cours d'exécution du programme

ALGOL, on crée des mémoires de manœuvre qui sont d'abord vides et « se remplissent » au fur et à mesure que s'exécutent les procédures d'entrée et les instructions d'affectation. Les procédures de sortie délivrent l'information à partir de la mémoire intérieure dans les canaux d'état de mémoire extérieure. Après la fin du processus algorithmique, la mémoire intérieure redevient vide, et le résultat d'exécution du programme ALGOL représente un état de mémoire extérieure.

§ 7.10. Remarques en conclusion

Un examen minutieux de l'ALGOL-60 amène à la conclusion que ce langage est loin d'être à la portée de tous. Son assimilation n'est pas facile et demande d'efforts considérables.

Il serait erroné de considérer l'ALGOL-60 comme un langage proche du langage usité des descriptions mathématiques. Le fait qu'il contient, en tant que symboles de base, des mots d'une langue naturelle (de l'anglais) ne rapproche pas l'ALGOL du langage de descriptions mathématiques. On ne peut pas non plus admettre que l'ALGOL-60 est un langage d'opérateur. Non seulement que les programmes ALGOL contiennent, en plus des ordres (instructions), des déclarations de type de variables, mais les ordres ALGOL eux-mêmes ne sont pas opérateurs au sens ordinaire du mot, puisque non connexes. Le calcul d'une expression faisant partie d'une instruction d'affectation, de branchement, de procédure, IF, alternative d'un aiguillage ou apparaissant dans l'en-tête d'une boucle, peut s'interrompre pour l'exécution d'une série d'instructions formant le noyau d'une procédure-fonction, et c'est seulement après qu'il peut être repris « à partir de l'endroit d'interruption ». De plus, étant donné l'éventualité de l'effet de bord, les valeurs de variables (l'état de mémoire) peuvent changer. Sans l'effet de bord, on pourrait ne pas considérer l'appel d'une procédure-fonction comme l'interruption du calcul d'une expression, et l'objection évoquée s'effacerait. Une instruction composée de l'ALGOL n'est pas, au fond, un opérateur non plus.

Il est intéressant de remarquer que, bien que l'ALGOL-60 soit décrit dans [1] à l'aide de formules métalinguistiques de Backus, donc aurait dû être indépendant du contexte (free contexte language), il n'en est pas ainsi en vérité (voir [9]). Cet état de choses est dû, à ce qu'il paraît, au fait que certaines règles de construction des textes de l'ALGOL, qui sont formulées d'une manière non formelle, ne peuvent pas être représentées au moyen de formules de Backus.

La notion la plus originale, nouvelle et très fructueuse fut celle de procédure. Les centres de calcul où l'on programme en ALGOL peuvent se communiquer non pas des programmes ALGOL complets, mais des descriptions de procédures. En effet, il est pratiquement

impossible d'incorporer des programmes ALGOL tout faits dans d'autres programmes, tandis que les descriptions de procédures sont spécialement destinées à être insérées. Pourtant, pour une bibliothèque de procédures conviennent seules les descriptions de procédures qui n'utilisent pas de variables non locales ni instructions de branchement renvoyant à l'extérieur du corps de procédure, de sorte que l'effet de bord n'a pas lieu. Notons que toutes les procédures standard (fonctions ou non) décrites en 7.7.9 et au § 7.5 doivent satisfaire aux conditions formulées.

Remarquons que certaines versions de l'ALGOL-60 sont assez répandues en U.R.S.S. Pourtant, actuellement, de plus en plus forte devient la tendance de passer de l'ALGOL au FORTRAN qui est plus simple, souple et non moins « puissant ».

Dirons en conclusion que, malgré ses inconvénients, l'ALGOL-60 présente néanmoins un grand intérêt. Ce langage algorithmique a beaucoup influencé le développement des idées dans le domaine de la programmation et, si l'on peut dire, « n'a pas encore dit son dernier mot ».

Faisons maintenant quelques mentions concernant le p. 7.7.2. La description du langage de référence ne dit pas comment sera exécuté un bloc apparaissant dans un corps de procédure et contenant un déclarateur || **own** ||, si les appels de la procédure viennent de différents endroits du programme ALGOL. Précisément, il n'est pas clair si dans un second appel le bloc est considéré le même que la première fois ou modifié (voir [1]). L'effet de déclarateur || **own** || dans les corps de procédures récursives est encore moins clair. Ne voulant pas reviser ce point délicat, les auteurs ont admis que le déclarateur || **own** || ne s'utilise pas dans les blocs appartenant aux corps de procédures.

Les auteurs rappellent que la présente description de l'ALGOL ne remplace pas la description officielle (par exemple, [1]), mais vise à aider à mieux comprendre celle-ci dans certains cas.

Dans le p. 35° de l'algorithme d'exécution du programme ALGOL figure la chaîne A qui se compose des numéros de points de l'algorithme à partir desquels il y a branchement vers le p. 35°. Avant le premier appel du p. 35° cette chaîne est vide. Plusieurs numéros de points peuvent s'y voir inscrits, car au cours du calcul de la valeur d'une fonction, plusieurs appels du p. 35° peuvent avoir lieu pour le calcul de « fonctions intérieures ». Or, les éléments portés à la chaîne A sont barrés lors de l'exécution du p. 32° (après avoir servi à la détermination de ξ, c'est-à-dire du numéro du point auquel il faut revenir après le p. 33°). Vers la fin du calcul de la valeur de la fonction qui a nécessité l'appel du p. 35°, la chaîne A redevient vide.

CHAPITRE 8

INTRODUCTION AU LANGAGE FORTRAN

§ 8.1. Alphabet du langage FORTRAN

Les symboles de base du langage FORTRAN qui forment son alphabet sont:

— les chiffres: || 0 || 1 || 2 || 3 || 4 || 5 || 6 || 7 || 8 || 9 ||;

— les lettres latines majuscules et minuscules;

— les valeurs logiques || .TRUE. || et || .FALSE .|| (chacun de ces mots, avec les points, représente un symbole);

— les limiteurs qui sont les signes d'opérations, les séparateurs et les descripteurs;

— le symbole monétaire $ (dollar);

— les noms d'instructions.

Les signes d'opérations sont:

— les signes d'opérations arithmétiques || + || — || * || / || ** || exprimant l'addition, la soustraction, la multiplication, la division et l'exponentiation;

— les signes de relations || .GT. || .GE. || .LT. || .LE. || .EQ. || .NE. || qu'on lit respectivement comme « supérieur à », « supérieur ou égal à », « inférieur à », « inférieur ou égal à », « égal à », « non égal à »;

— les signes d'opérations logiques || .OR. || .AND. || .NOT. || qui désignent « disjonction logique », « conjonction logique », « négation logique ».

Les séparateurs sont:

— les séparateurs constructifs || , || . || = || ' || qu'on appelle respectivement virgule, point décimal, égalité *), apostrophe;

— les parenthèses gauche || (|| et droite ||) ||;

— les noms d'instructions séquentielles || GO TO || IF || DO || CONTINUE || PAUSE || RETURN || STOP || qui sont des symboles indivisibles et se lisent respectivement comme « aller à », « si », « boucle », « continuer », « arrêter », « retourner », « arrêt ».

Les descripteurs sont || INTEGER || REAL || COMPLEX || LOGICAL || DEFINE FILE || et représentent des symboles indivisibles qui se lisent comme « entier », « réel », « complexe », « logique », « description de fichier ».

---

*) Il joue le rôle du signe d'affectation dans les expressions arithmétiques.

On utilise en FORTRAN un grand nombre de mots réservés qui sont des symboles indivisibles. Ce sont, par exemple, les noms d'instructions.

§ 8.2. Structures primaires en FORTRAN

Les structures primaires en FORTRAN sont les nombres (on les appelle également constantes numériques), les identificateurs et les chaînes de caractères.

Toute suite finie de chiffres s'appelle *nombre entier sans signe* (ou *entier sans signe* tout court). Une structure représentant un entier sans signe ou un entier sans signe précédé d'un || + || ou d'un || — || s'appelle *nombre entier.*

EXEMPLE 8.1. Les structures

|| 640 || 0700 || 0 || 00 || 000 || 156 || 78123 ||

sont des entiers sans signe;

|| 0 || —0 || +0 || —00 || 00 || 17 || +1764 || 005604 ||

sont des entiers.

On distingue les nombres entiers de longueur standard et de longueur non standard. Cette distinction est liée à la représentation des entiers dans la mémoire de machine. Ainsi, dans les machines de troisième génération la longueur non standard des entiers est égale à 2 octets (16 positions binaires dont l'une de signe), et la longueur standard est de 4 octets (32 positions binaires dont l'une de signe). D'où la définition suivante.

Un nombre entier s'appelle *entier de longueur non standard* si son module appartient à l'intervalle de 0 à $2^{15} - 1$, et *entier de longueur standard* si son module appartient à l'intervalle de 2^{15} à $2^{31} - 1$.

EXEMPLE 8.2. Les structures

|| —32700 || —0601 || || 0 || +00 || 276 ||

sont des entiers de longueur non standard, et les structures

|| 40001 || +1762501 || —560064 || +104027 || —20765345 ||

sont des entiers de longueur standard.

On appelle *fraction régulière* un entier sans signe précédé d'un point décimal.

On appelle *nombre réel sans exposant* soit un entier suivi d'un point, soit une fraction régulière, soit une fraction régulière précé-

dée d'un signe || + || ou || — ||, soit une chaîne représentant un entier suivi d'une fraction régulière.

EXEMPLE 8.3. Les notations suivantes sont des fractions régulières :

|| .00 || .164 || .016 ||;

tandis que les notations

|| 04. || + 6075. || — 700. || .015 || + .117 || — .080 || + 06.25 || 360.00 || — 18.276 ||

sont des nombres réels sans exposant.

Les notions de longueurs standard et non standard pour les nombres réels sont également liées à la représentation des réels dans la machine. Les réels de longueur standard occupent, dans la mémoire de machine de troisième génération, 4 octets, la longueur non standard étant égale à 8 octets. D'où la définition suivante.

Un nombre réel sans exposant est dit *de longueur standard* ou *en simple précision* s'il contient sept chiffres au plus; dans le cas contraire il est dit *de longueur non standard* ou *en double précision.*

EXEMPLE 8.4. Les notations

|| — .01640 || 16.2300 || — 0.000||

sont des nombres réels sans exposant de longueur standard; les notations

|| 7620.1891 || + 0.1604201028 ||

étant des réels sans exposant en double précision.

On appelle *exposant* (*décimal*) une chaîne formée d'un séparateur || E || ou || D || et d'un entier.

EXEMPLE 8.5. Les notations suivantes sont des exposants :

|| E23 || E — 07 || E + 3 || E — 2 || D00 || D + 4 ||.

On appelle *nombre réel avec exposant* un nombre réel sans exposant suivi d'un exposant.

Un nombre réel avec le symbole || E || pour séparateur est considéré comme *réel avec exposant en simple précision*; un nombre réel avec le symbole || D || pour séparateur est considéré comme *réel avec exposant en double précision.*

EXEMPLE 8.6. Nombres réels avec exposant en simple précision :

|| — .27E 2 || 64.02E — 12 || + 0.01E — 6 ||;

nombres réels avec exposant en double précision :

|| 0.16D — 10 || — 105.45D 04 || 0.D + 2 ||.

EXEMPLE 8.7. Au nombre FORTRAN || — .2716E5 || il correspond en représentation usuelle le nombre décimal $—0{,}2716 \cdot 10^{5}$.

Au nombre FORTRAN || 67.25D — 04 || il correspond en représentation usuelle le nombre décimal $67,\ 25 \cdot 10^{-4}$.

On appelle *nombre complexe* un couple de nombres réels qui sont écrits successivement sur une ligne, séparés par une virgule et mis entre parenthèses. Le premier de ces nombres réels forme la partie réelle du nombre complexe, le deuxième est la partie imaginaire.

EXEMPLE 8.8. Exemples de nombres complexes :

|| (3.07623, — .19E04) || (6.D — 2,0.3E8) || (.24, — 167.25034) ||

(1.05E — 4, — 0.64E5) || (99.0D — 2,— .4D06)||.

De même que les nombres réels, les nombres complexes peuvent être en simple ou double précision.

Il s'agit d'un nombre complexe simple précision, lorsque les deux nombres réels qui en font partie sont en simple précision, sinon le nombre complexe est en double précision.

EXEMPLE 8.9. Au FORTRAN-nombre complexe || (— .6403,24.E— —3) || il correspond, dans la notation usuelle, le nombre complexe $— 0{,}6403 + 24 \cdot 10^{-3}\ i$.

On appelle *identificateur* (nom symbolique) une suite de 1 à 7 caractères alphanumériques ; le premier devant être une lettre.

EXEMPLE 8.10. Les notations || A || A7B || X23 || ALPHA || EF306A4 || sont des identificateurs. Les notations || 125A7 || (commence par un chiffre) et || A2BCDE76 || (contient huit caractères) ne sont pas des identificateurs.

Chaîne de caractères. Une suite quelconque de symboles s'appelle *corps de chaîne*. On appelle *chaîne* soit un corps de chaîne mis entre guillemets, à condition que chaque guillemet dans le corps de chaîne soit remplacé par un couple de guillemets, soit une suite de trois éléments dont le premier est un entier non signé, le deuxième la lettre H, le troisième un corps de chaîne, le premier élément (l'entier non signé) indiquant le nombre de caractères dans le corps de chaîne. Dans le premier cas la chaîne est appelée *chaîne sans indication de longueur*, dans le deuxième *chaîne avec indication de longueur*.

Exemple 8.11. Les notations || A5, K + 5 — 1 || Z23 * BJ || 127'A'L || sont des corps de chaîne.

Les notations || 'A5, K + 5 — 1' || 'Z23 * BJ' || '127"A"L' || sont des chaînes sans indication de longueur; les notations || 8HA5, K + 5 — 1 || 6H Z23 * BJ || 7H 127'A'L || sont des chaînes avec indication de longueur.

§ 8.3. Variables. Tableaux

On distingue des variables simples et des variables indicées.

Une *variable simple* représente une quantité qui prend des valeurs numériques ou logiques et qui est désignée par un identificateur.

On appelle *tableau* un ensemble fini de données identifiées par un même nom symbolique et ordonnées suivant les valeurs d'un système de paramètres entiers allant chacun de l'unité jusqu'à sa valeur maximale. L'élément d'un tableau s'appelle *variable indicée*. L'identificateur commun de tous les éléments de tableau s'appelle *identificateur de tableau*. Un tableau est noté par un identificateur de tableau suivi d'une liste, entre parenthèses, des indices dont le total ne dépasse pas sept. Chaque indice est représenté par une expression d'indice qui peut avoir l'une des forme suivantes: $a * x + b$, $a * x - b$, $a * x$, $x + b$, $x - b$, x, b, où a et b sont des entiers non signés, x une variable entière. Dans ces expressions, les symboles || + || — || * || sont respectivement les signes d'addition, de soustraction et de multiplication (pour plus de détail sur ces expressions, voir le § 8.5).

Exemple 8.12. Exemples de variables simples || J || A21 || IKS || F560SOS ||.

Des tableaux peuvent avoir pour désignations: || RUSS (X) || A (5 * B — 6, X) || A3 (W + 1, Z — 2, 4) ||.

Les variables indicées de ces tableaux peuvent être notées comme suit:

|| RUSSE (2) || A (4, 6) || A3 (1, 7, 4) ||.

Le nombre d'indices dans la description d'un tableau s'appelle *dimension du tableau*.

Dans le FORTRAN, il est convenu d'ordonner les éléments d'un tableau k-dimensionnel A de la manière suivante:

$$A(1, 1, \ldots, 1),\ A(2, 1, \ldots, 1),\ \ldots,\ A(i_1, 1, \ldots, 1),$$

$$A(1, 2, \ldots, 1),\ A(2, 2, \ldots, 1),\ \ldots,\ A(i_1, 2, \ldots, 1),$$

$$A(1, i_2, \ldots, i_k),\ A(2, i_2, \ldots, i_k),\ \ldots,\ A(i_1, i_2, \ldots, i_k).$$

Les variables sont classées en quatre types: entières, réelles, complexes et logiques. Une variable peut prendre des valeurs différentes mais correspondant à son type. De plus, les variables entières, réelles et complexes peuvent être en simple ou en double précision.

Le type de variables est déclaré de trois façons. La première consiste à choisir, pour déclarer une variable entière de longueur standard, un nom qui commence par l'un des symboles || I || J || K || L || M || N ||. Pour désigner une variable réelle simple précision, on utilise en tant que première lettre du nom de la variable n'importe quelle lettre latine différente des lettres indiquées.

EXEMPLE 8.13. Les variables suivantes sont des variables entières de longueur standard || I ||LIST (2)|| KOL|| INDEX|| M24A||.

Exemples de variables réeles simple précision: || X|| VAL (3, 7)|| Y|| AB 102|| COC||.

Le deuxième mode de déclaration du type de variable représente l'utilisation d'une instruction de déclaration de type implicite, qui donne le type d'une variable (|| INTEGER|| REAL|| COMPLEX|| LOGICAL||) par le premier symbole qui figure dans le nom de celle-ci. L'instruction permet d'indiquer également les longueurs de variables (voir p. 8.9.3).

La troisième méthode de déclaration de type est l'utilisation d'une instruction explicite. Contrairement à une instruction implicite, elle indique le type d'une variable non pas par le premier caractère de son nom, mais par son nom complet (voir p. 8.9.3).

Pour décrire un tableau et la portée de ses indices, on emploie une instruction de déclaration de dimensions (voir p. 8.9.3).

§ 8.4. Identificateurs de fonctions

En plus des signes d'opérations qui sont des symboles de base, on emploie dans le FORTRAN des signes d'opérations plus compliqués que l'on appelle *identificateurs* (*références*) *de fonctions*. L'exécution d'opérations données par une référence de fonction se ramène à l'exécution de l'algorithme correspondant. Les arguments d'une fonction peuvent être les nombres, les variables, les expressions (arithmétiques ou logiques), les identificateurs de tableaux, de fonctions et de procédures.

Une référence de fonction est constituée d'un nom symbolique (nom de la fonction) suivi d'une liste de paramètres effectifs (ou arguments) séparés par des virgules, entre parenthèses.

Les valeurs d'une fonction peuvent être de type entier, réel, complexe ou logique. Certaines fonctions très usitées possèdent en FORTRAN des désignations fixes:

|| SQRT (A)|| — racine carrée de A;

|| SIN (A) || COS (A) || TANH (A) || — fonctions sin A, cos A, th A ||

|| ATAN (A) || — fonction arc tg A ;

|| EXP (A) || — fonction e^A ;

|| ALOG (A) || — fonction Ln A ;

|| ALOG10 (A) || — fonction log A.

Chacune de ces fonctions définit une variable réelle simple précision.

La notation Dx, où x est la désignation d'une des fonctions standard, définit une variable réelle double précision.

§ 8.5. Expressions

8.5.1. Expression logique. La définition d'une expression logique (booléenne) est récursive.

On appelle *primaires logiques* les valeurs logiques, les variables logiques, les identificateurs de fonctions logiques, les relations, de même que les expressions logiques entre parenthèses.

On appelle *relation* tout couple d'expressions arithmétiques (voir p. 8.5.2), en particulier de nombres et d'identificateurs de variables numériques, reliées par l'un des signes d'opération de relation : || .GT. || .GE. || .LT. || .LE. || .EQ. || .NE. ||. Une relation prend la valeur « vrai » si elle est satisfaite pour toutes les expressions arithmétiques y figurant, elle prend la valeur « faux » dans le cas contraire.

On appelle *secondaire logique* un primaire logique précédé ou non de l'opérateur de négation || .NOT. || (non).

On appelle *expression logique* soit un secondaire logique soit une suite constituée d'une expression logique, de l'un des symboles || .AND. || .OR. || (et, ou) et d'un secondaire logique. En d'autres termes, une expression logique représente une suite finie de secondaires logiques reliés par les symboles || .AND. || .OR. ||.

Exemple 8.14. Les notations suivantes sont des expressions logiques : || .TRUE. || A2. LT. 2.3576 || .NOT. MIR (X, Y) || B .OR. B2A .AND. XB21 .AND. .NOT. ZET (U) || A .AND. ((A .OR. B2) .AND. .TRUE.) ||.

Les signes d'opérations de relation et d'opérations logiques ont la signification usuelle.

En évaluant la valeur d'une expression logique on effectue les opérations logiques de gauche à droite, dans l'ordre de hiérarchie usuel d'opérateurs et de parenthèses.

Rappelons l'hiérarchie d'opérations, dans l'ordre décroissant : 1) évaluation des expressions arithmétiques ; 2) évaluation des rela-

tions; 3) négation logique; 4) multiplication logique; 5) addition logique.

8.5.2. Expression arithmétique. La définition de l'expression arithmétique est récursive et analogue à celle de l'expression logique.

On appelle *primaires arithmétiques* les nombres, les variables, les identificateurs de fonctions de type entier, réel ou complexe, de même que les expressions arithmétiques (quelconques) entre parenthèses.

On appelle *facteur* soit un primaire arithmétique, soit une suite composée d'un facteur, d'un signe || ** || et d'un primaire arithmétique.

On appelle *terme* soit un facteur isolé, soit une suite composée d'un terme, de l'un des signes || * || et || / || et d'un facteur.

On appelle *expression arithmétique* un terme précédé ou non de l'un des signes || + || et || — ||, ainsi que toute suite formée d'un primaire arithmétique, de l'un des signes || + || et || — || et d'un terme.

EXEMPLE 8.15. Voici quelques expressions arithmétiques: || 25 || X || A (3) || X2 — 64.8 * IVAN + X**3**Y || SQRT (X (2)) + B — R * Z**2 || E (X, Y) + X*(A + X*(B — — X * (A + D — C))) ||.

En évaluant la valeur d'une expression arithmétique on tient compte des règles communément admises d'utilisation de parenthèses et de hiérarchie d'opératuers. Voici l'ordre hiérarchique décroissant d'opérations: 1) évaluation des valeurs des identificateurs de fonctions; 2) || ** ||; 3) || * || / ||; 4) || + || — ||.

Dans le cas d'opérateurs de hiérarchie égale (à l'exception de || ** ||) on les effectuera de la gauche vers la droite.

EXEMPLE 8.16. Dans l'expression arithmétique || X*B2/C * X—A ||, l'ordre d'exécution est:

1) X * B2
2) (X * B2)/C
3) ((X * B2)/C) * X
4) (((X * B2)/C) * X) — A)

Les opérations d'élévation à une puissance, lorsqu'elles voisinent dans une expression, sont effectuées de la droite vers la gauche.

EXEMPLE 8.17. Dans l'expression arithmétique || X**Y** Z**X1 || l'ordre d'exécution est:

1) Z**X1
2) Y**(Z**X1)
3) X**(Y**(Z**X1))

Le type de chaque expression arithmétique de la forme || A + + B || A — B || A * B || A/B || est déterminé par l'ensemble des règles de la table 8.1. On y adopte les désignations ES, EN, RS, RD, CS,

Table 8.1

Détermination du type d'une expression arithmétique AσB, où σ est un des signes || + || — || * || / ||

B / A	ES	EN	RS	RD	CS	CD
ES	ES	ES	RS	RD	CS	CD
EN	ES	ES	RS	RD	CS	CD
RS	RS	RS	RS	RD	CS	CD
RD	RD	RD	RD	RD	CD	CD
CS	CS	CS	CS	CD	CS	CD
CD	CD	CD	CD	CD	CD	CD

CD pour les types respectifs: entier longueur standard, entier longueur non standard, réel simple précision, réel double précision, complexe simple précision, complexe double précision.

Le type d'une expression arithmétique de la forme || A ** B || est déterminé en conformité de la table 8.2, sauf le cas où A < 0 et B est une quantité réelle. Dans ce cas le résultat de l'opération || A * * B || est indéterminé.

Table 8.2

Détermination du type d'une expression arithmétique || A ** B ||

B / A	ES	EN	RS	RD	CS	CD
ES	ES	ES	RS	RD	×	×
EN	ES	EN	RS	RD	×	×
RS	RS	RS	RS	RD	×	×
RD	RD	RD	RD	RD	×	×
CS	CS	CS	×	×	×	×
CD	CD	CD	×	×	×	×

Dans la table 8.2. le symbole × exprime une indétermination, les autres désignations étant les mêmes que dans la table 8.1.

Les signes des opérations arithmétiques dans le FORTRAN ont le sens usuel, à l'exceptions des cas suivants:

1. Le quotient d'un entier par un entier est la partie entière du quotient véritable.

2. Le résultat de l'élévation à une puissance entière négative d'une base entière est la partie entière de la puissance.

3. Le résultat d'une addition (soustraction, multiplication, division) dont l'un des opérandes est du type complexe et l'autre du type entier (réel), est un nombre complexe dont la partie réelle est la valeur de l'opérande entier (réel), et dont la partie imaginaire est nulle.

§ 8.6. Instructions

Les principaux éléments au moyen desquels sont donnés les algorithmes en FORTRAN sont les instructions. Les expressions décrites plus haut représentent des parties constituantes d'instructions. Une instruction peut être étiquétée ou non. Si N est une instruction non étiquetée et M est une étiquette, alors || N || MN || sont des instructions.

On distingue dans le FORTRAN les instructions des types suivants: instructions inconditionnelles, instructions conditionnelles, instruction d'arrêt, instructions d'entrée-sortie.

Les instructions inconditionnelles se classent à leur tour en *instructions d'affectation*, *instructions de branchement*, *vides*, *instructions de procédures*.

Les instructions conditionnelles sont représentées par l'*instruction* IF et l'*instruction de boucle*.

Il existe deux types d'instructions d'entrée-sortie: *instructions d'entrée-sortie à accès séquentiel* et *instruction d'entrée-sortie à accès direct*.

En plus des instructions énumérées, que l'on peut considérer comme principales, le FORTRAN possède des instructions dites non exécutables parmi lesquelles se classent les *instructions de définition de formats*, *de procédures*, *de fonctions*, *de description des types de grandeurs*, *d'organisation de données* (COMMON), *d'organisation de mémoire* (EQUIVALENCE), etc.

8.6.1. Etiquette. Nous avons déjà dit que les instructions d un algorithme peuvent être étiquetées. Une *étiquette* est un nombre entier non signé de 5 chiffres au plus.

8.6.2. Instructions d'affectation. On considère les instructions d'affectation de deux espèces.

On appelle *instruction d'affectation de première espèce* toute structure de la forme

$$x = w,$$

où x est une variable (la partie gauche de l'instruction d'affectation), w une expression arithmétique ou logique. L'instruction d'affectation est dite *arithmétique* ou *logique*, selon que l'expression w est arithmétique ou logique. Le signe || =|| est appelé *signe d'affectation.* Il veut dire qu'il faut donner à la quantité située à gauche la valeur de l'expression écrite à droite du signe d'affectation.

Les parties gauche et droite d'une instruction d'affectation logique sont du type logique ; pour une instruction d'affectation arithmétique elles sont d'un type arithmétique (entier, réel ou complexe). Lorsque le type et la taille de la partie droite d'une instruction d'affectation arithmétique diffèrent du type et de la taille de sa partie gauche, la valeur de la partie droite est automatiquement transformée et devient de type et de taille demandés par la partie gauche. Ces transformations sont conformes aux conventions suivantes :

1. Si x est entier et w du type réel, on donne à x une valeur égale à la partie entière de w.
2. Si x est du type entier et w du type complexe, on donne à x une valeur égale à la partie entière de la composante réelle de w.
3. Si x est du type réel et w du type complexe, on donne à x une valeur égale à la partie réelle de w.
4. Si x est du type complexe et w du type entier ou réel, la partie réelle de x sera la valeur de w, et la partie imaginaire, nulle.

On appelle *instruction d'affectation de deuxième espèce* (ou instruction ASSIGN) une structure de la forme

|| ASSIGN n TO m||,

où n est une étiquette d'instruction, m une variable de type entier.

Cette instruction donne à la variable m la valeur de l'étiquette n. C'est pour cette raison que l'instruction d'affectation de deuxième espèce est appelée *instruction d'affectation d'étiquette.* Remarquons que la variable m qui a pris la valeur d'une étiquette ne peut être utilisée ni comme variable à valeur numérique avant qu'une telle valeur ne lui soit donnée à l'aide d'une instruction d'affectation arithmétique ou d'une instruction d'entrée, ni dans une instruction GO TO imposé sauf si elle a été redéfinie.

8.6.3. Instructions de branchement. On distingue dans les instructions de contrôle trois types d'instructions de branchement inconditionnel : GO TO inconditionnel, GO TO imposé, GO TO calculé.

Une *instruction* GO TO *inconditionnel* a la forme || GO TO n||, où n est une étiquette.

L'exécution de cette instruction a pour effet de prendre comme instruction suivante celle indiquée par d'étiquette n.

EXEMPLE 8.18. L'instruction || GO TO 101|| est une instruction de branchement inconditionnel dont l'exécution provoque toujours le transfert du contrôle du programme à l'instruction d'étiquette 101.

Une *instruction* GO TO *imposé* a la forme

$$|| \text{GO TO } x, (n_1, n_2, \ldots, n_k)||,$$

où x est un nom de variable entière simple, n_i ($i = 1, \ldots, k$) sont des étiquettes d'instructions.

L'exécution de cette instruction provoque le transfert du contrôle à l'instruction d'étiquette n_i qui coïncide avec la valeur de la variable x que cette dernière doit avoir reçue vers le moment d'exécution de l'instruction donnée (GO TO imposé) par l'exécution préalable d'une instruction d'affectation d'étiquette.

EXEMPLE 8.19. L'instruction || GO TO A2B, (10, 51, 37, 0002)|| est une instruction de branchement imposé. Elle passe le contrôle à l'instruction étiquette 37 à condition que, vers le moment de son exécution, la variable A2B ait reçu la valeur 37.

Une *instruction* GO TO *calculé* a la forme

$$|| \text{GO TO } (n_1, n_2, \ldots, n_k), x||,$$

où n_i ($i = 1, 2, \ldots, k$) sont des étiquettes d'instructions, x est le nom de variable entière simple.

L'exécution de cette instruction a pour effet de prendre comme instruction suivante celle identifiée par l'étiquette n_j, où j est la valeur de x au moment de l'exécution et $1 \leqslant j \leqslant k$. Si à ce moment la variable x prend une valeur supérieure à k, le contrôle passe à l'instruction immédiatement suivante.

8.6.4. Instructions de comparaison. On distingue deux types de comparaisons (instructions IF) : *test arithmétique* et *test logique*.

Le test arithmétique a la forme

$$|| \text{IF } (w)\ n_1, n_2, n_3||,$$

où w est une expression arithmétique de type entier ou réel, n_i ($i = 1, 2, 3$) sont trois étiquettes d'instruction.

Cette instruction calcule la valeur de l'expression arithmétique w et passe le contrôle à l'instruction étiquetée n_1 (n_2, n_3) selon que la valeur de l'expression w est inférieure (égale, supérieure) à zéro.

EXEMPLE 8.20. L'instruction || IF (A + B) 0100, 22, 15|| est un test arithmétique. Elle passe le contrôle à l'instruction étiquetée 15, si vers le moment de son exécution A = 24.01 et B = 0.2.

L'instruction || IF (*w*)|| dans une construction || IF (*w*) *Q*||, où *w* est une expression logique, *Q* une instruction, s'appelle *test logique*. *Q* est n'importe quelle instruction exécutable sauf une instruction de boucle (voir p. 8.6.7) ou une autre instruction IF logique.

Cette instruction évalue la valeur de l'expression *w* et passe le contrôle à l'instruction *Q* si la valeur de *w* est || .TRUE.||; si la valeur de *w* est || .FALSE.||, le programme exécute la première instruction qui suit.

EXEMPLE 8.21. Soit

```
|| IF (X.LE.0.0) A = 2.307 — B * A
   A1 — A ** 2||
```

un fragment de programme FORTRAN. Ici || IF (X.LE.0.0)|| est une instruction qui effectue un branchement à l'instruction arithmétique d'affectation || A = 2.307 — B * A|| si au moment de son exécution la valeur de l'expression logique || X.LE.0.0|| est || .TRUE.||, i.e. si la valeur de *X* est inférieure ou égale à 0.0; la commande passe à l'instruction arithmétique d'affectation || A1 = = A ** 2|| si la valeur de l'expression logique || X.LE.0.0|| est || .FALSE.||, i.e. si la valeur de *X* est supérieure à 0.0.

8.6.5. Instruction vide. On appelle ainsi la structure

|| CONTINUE ||.

Cette instruction est ineffective: le contrôle du programme passe à la prochaine instruction.

8.6.6. Instructions d'arrêt. Il y a deux instructions d'arrêt: une instruction de la forme

|| PAUSE *n*||,

et une instruction de la forme

|| STOP *n*||,

où *n* est soit vide, soit un entier non signé.

Ces deux instructions provoquent un arrêt de l'exécution du programme et fournissent au pupitre de commande le nombre *n*.

8.6.7. Instruction de boucle. Une instruction de boucle a la forme

|| DO $n\ x = m_1,\ m_2,\ m_3$||

où || DO || définit une boucle, n est une étiquette, x est une variable entière non indicée appelée *variable de contrôle*, et les éléments m_1 (*valeur initiale*), m_2 (*valeur finale*) et m_3 (*pas de progression ou incrément*) sont chacun soit une constante entière non signée, soit une variable entière non indicée. Si m_3 n'est pas indiqué, alors $m_3 = 1$ et l'instruction de boucle prend la forme

$$\| \text{DO } nx = m_1, m_2 \|.$$

Vers le moment d'exécution de l'instruction DO, les valeurs initiale et finale de la variable de contrôle, de même que l'incrément doivent avoir des valeurs positives.

On entend par boucle une suite d'instructions comprises entre la première instruction, appelée *instruction initiale* qui suit immédiatement l'instruction DO, et la dernière, appelée *instruction terminale* et étiquetée n. Il est défendu d'utiliser en tant qu'instruction terminale une instruction GO TO, un IF arithmétique, une instruction de retour, une instruction d'arrêt, de même qu'un IF logique contenant l'une des instructions énumérées, si l'instruction qui suit immédiatement l'instruction terminale n'est pas || CONTINUE ||.

Une instruction DO commande l'exécution d'une boucle. Ceci est réalisé de la manière suivante: on commence par affecter à la variable de contrôle la valeur représentée par le paramètre initial et le contrôle passe à l'instruction initiale de la boucle; après l'exécution de l'instruction terminale la variable de contrôle est incrémentée de m_3 et si la valeur obtenue ne dépasse pas la valeur finale, la commande revient à l'instruction initiale de la boucle et ainsi de suite. Le processus d'exécution de la boucle se termine dès que la valeur de la variable de contrôle dépasse sa valeur finale, auquel cas la commande passe à la première instruction qui suit l'instruction terminale. De la sorte, le nombre d'exécutions de la boucle vaut $[(m_2 - m_1)/m_3] + 1$, où $[(m_2 - m_1)/m_3]$ est la partie entière du nombre $(m_2 - m_1)/m_3$, si $m_2 > m_1$. Et si $m_2 < m_1$, la boucle s'exécute une seule fois.

EXEMPLE 8.22. Soit un fragment de programme FORTRAN:

```
|| DO 17 M = 1, 100
X = 4.3 + X * Y ** 2
B = X - 3
 17INDEX (I) = INDEX (I) - X
A = B + C||
```

Ici l'instruction de boucle est exprimée par la structure || DO 17 M = 1, 100||, la boucle est constituée par les instructions d'affectation qui calculent les quantité X, B et INDEX. La pre-

mière de ces instructions est l'instruction initiale, la dernière, l'instruction terminale. M est la variable de contrôle, sa valeur initiale vaut l'unité, sa valeur finale vaut cent, l'incrément est l'unité.

Au début la variable de contrôle prend la valeur unité. Les instructions de la boucle s'exécutent une fois, et la valeur de la variable de contrôle s'incrémente d'une unitié. Les instructions de la boucle s'exécutent une nouvelle fois, etc. Après cent itérations le paramètre devient égal au nombre 101, et la commande passe à l'instruction calculant la quantité A.

En écrivant une boucle il faut avoir en vue que:

1. Les instructions d'une boucle ne doivent pas modifier les valeurs des quantités figurant dans le nom de la boucle.
2. Un branchement de l'extérieur à une instruction de boucle n'est possible que si un branchement préalable a eu lieu de l'intérieur de la boucle vers l'extérieur et si les paramètres de l'instruction DO n'ont pas été modifiés à l'extérieur.
3. Une boucle peut contenir une instruction d'appel d'une fonction ou d'un sous-programme.
4. Une boucle peut contenir à l'intérieur une autre boucle, mais les « intersections » de boucles ne sont pas admises.

Exemple 8.23. Soient deux boucles emboîtées:

```
|| DO 17 K = 1, 11, 2
     10 A = A + 2
   DO 17 L = 1, 20
     11 X = X + X ** 2
     17 Y = Y + (A — X)/B ||.
```

Ce fragment de programme peut être réalisé à l'aide d'autres instructions FORTRAN comme suit:

```
|| K = 1
10A = A + 2
   L = 1
11X = X + X ** 2
17Y = Y + (A — X)/B
IF (L.LE.20) GO TO 11
   K = K + 1
IF (K .LE. 11) GO TO 10||.
```

8.6.8. Instruction d'appel d'un sous-programme. Elle se compose d'un nom d'instruction || CALL ||, d'un identificateur de sous-

programme et, éventuellement, d'une liste des paramètres effectifs (séparés par des virgules et mis entre parenthèses), c'est-à-dire que la forme générale d'une instruction CALL est

$$\| \text{CALL } p\ (x_1,\ x_2,\ \ldots,\ x_n) \|,$$

où p est un identificateur de sous-programme, x_i $(i = 1, 2, \ldots, n)$ sont des paramètres effectifs. En tant que paramètres effectifs d'une procédure on peut employer nombres, variables, expressions arithmétiques et logiques, identificateurs de fonctions et de sous-programmes, étiquettes d'instructions.

Cette instruction constitue l'appel du sous-programme dont le nom est p. Elle effectue l'appel du sous-programme, remplace ses paramètres formels par les paramètres effectifs (pour les sous-programmes à paramètres), passe la commande à la première instruction exécutable du sous-programme et range l'adresse de retour au programme principal. Après l'exécution du sous-programme le contrôle passe soit à l'instruction qui suit l'instruction CALL, soit à une instruction dont l'étiquette est contenue dans la liste des paramètres effectifs de l'instruction CALL. (Pour plus de détail, voir p. 8.9.4.3.)

§ 8.7. Fichiers

8.7.1. Fichier. Deux types de fichiers. Les instructions d'entrée-sortie réalisent l'échange d'informations entre la mémoire extérieure (bandes magnétiques, disques, imprimantes, etc.) et la mémoire principale d'un calculateur. L'information d'entrée ou de sortie est normalement mise sous forme d'une suite d'enregistrements qui s'appelle *fichier* (ou file). Chaque enregistrement représente une suite de symboles.

Un fichier a pour nom un nombre entier non signé ou une variable entière. (Dans le FORTRAN, on considère comme fichier une suite de données enregistrées sur bande magnétique, disque, tambour, bande de papier, jeu de cartes perforées, etc.) On distingue deux types de fichiers : à accès séquentiel et à accès direct. Un *fichier à accès séquentiel* est un fichier où sont définis un enregistrement initial, un enregistrement final et, pour tout autre enregistrement, les notions d'enregistrement « suivant » et d'enregistrement « précédent ». On utilise un fichier à accès séquentiel lorsque les enregistrements sont écrits (ou lus) dans le fichier dans l'ordre de leur succession.

Un *fichier à accès direct* est un fichier pour lequel la numérotation des enregistrements a une importance déterminante. On s'en sert lorsqu'il faut écrire (ou lire) des enregistrements dans le fichier dans un ordre arbitraire. Chaque fichier à accès direct doit être décrit dans le programme une fois, au moyen d'une instruction de description de fichier (voir p. 8.7.2.).

8.7.2. Instruction de description de fichier. Cette instruction a la forme

$$\|\text{DEFINE FILE } a_1\,(m_1,\ l_1,\ r_1,\ t_1),\ a_2\,(m_2,\ l_2,\ r_2,\ t_2),\ \ldots$$
$$\ldots,\ a_n\,(m_n,\ l_n,\ r_n,\ t_n)\|,$$

où a_i $(i = 1, 2, \ldots, n)$ sont des numéros de fichiers; m_i un entier non signé qui détermine le nombre des enregistrements du a_i-ème fichier; l_i un entier non signé qui détermine la taille maximale des enregistrements du a_i-ème fichier.

La quantité l_i est évaluée soit en symboles (octets), soit en mots (un mot vaut 4 symboles ou 4 octets). Le choix de l'unité de mesure est déterminé par le paramètre r_i qui peut prendre l'une des trois valeurs possibles: L, E ou U. Si r_i est L, la taille d'enregistrements dans les fichiers est donnée en octets, et l'appel d'une tel fichier est réalisé par une instruction d'entrée-sortie avec format (v.p. 8.8.3) et sans format (v.p. 8.8.2). Si r_i est E, la taille d'enregistrements de fichiers est donnée toujours en octets, mais l'appel d'un tel fichier n'est réalisé que par une instruction d'entrée-sortie avec format. Enfin, si r_i est U, la taille d'enregistrements de fichiers est mesurée en mots, et l'appel d'un fichier est réalisé par une instruction d'entrée-sortie sans format.

Le paramètre t_i est une variable entière simple. Après l'exécution d'une instruction d'entrée-sortie, elle prend automatiquement la valeur égale au numéro de l'enregistrement qui suit immédiatement le dernier enregistrement introduit. Après l'exécution de l'instruction de recherche d'élément de fichier, la variable prend la valeur égale au numéro de l'enregistrement trouvé.

Un programme FORTRAN (voir § 8.10) peut contenir plusieurs instructions de description de fichiers, mais seule la dernière d'entre elles est effective.

Exemple 8.24. La notation || DEFINE FILE 4 (30, 50, L, X), 6 (100, 50, L, Y2)|| est une instruction de description des fichiers de numéros 4 et 6. Le premier fichier se compose de 30 enregistrements de 50 octets chacun. La spécification L veut dire que l'information peut être entrée ou sortie avec ou sans format. Après l'exécution d'une instruction d'entrée-sortie, la variable de contrôle X prend la valeur égale au numéro de l'enregistrement qui suit l'enregistrement introduit.

Si la notation en question contenait le symbole E à la place de L, cela impliquerait l'utilisation des instructions d'entrée-sortie avec format.

Lorsque l'entrée-sortie est réalisée au moyen d'instructions sans format, la description de fichiers doit avoir la forme || DEFINE FILE 4 (30, 50, U, X), 6 (100, 50, U, Y2)||.

§ 8.8. Instruction d'entrée-sortie

8.8.1. Liste d'entrée-sortie. Aux instructions d'entrée-sortie est liée la notion de liste d'entrée-sortie qui détermine les zones de la mémoire principale que l'on utilise pour la transmission de l'information (voir, par exemple, p. 8.8.2.).

Une liste d'entrée-sortie a la forme $a_1, a_2, \ldots, a_n$ où a_i ($i = 1, 2, \ldots, n$) sont les éléments de la liste. Ce peuvent être les variables (simples ou indicées), les identificateurs (noms) de tableaux ou les éléments du type boucle (on dit encore liste avec DO implicite).

On entend ici par *élément boucle* une structure de la forme

$$(b_1, b_2, \ldots, b_k, x = m_1, m_2, m_3),$$

où b_j ($j = 1, 2, \ldots, k$) sont soit des variables, soit des éléments boucles, $x = m_1, m_2, m_3$ étant un nom de boucle (voir p. 8.6.7).

Lorsqu'un élément de liste représente une variable, cela veut dire qu'il faut transmettre les valeurs de cette variable. Lorsqu'il représente un identificateur de tableau, il s'agit de transmettre tous les éléments du tableau dans l'ordre de leur succession. Lorsqu'un élément est une liste avec DO implicite, il faut transmettre les valeurs des variables spécifiées dans la liste, dans l'ordre déterminé par la variation de x dans la boucle.

EXEMPLE 8.25. Soit || (X (J, I + 1), Y (I), I = 2, 9, 3)|| une liste d'entrée-sortie avec DO implicite. Cela signifie qu'il faut transmettre les valeurs des éléments des tableaux X et Y : X (J, 3), Y (2), X (J, 6), Y (5), X (J, 9), Y (8).

Si un élément boucle contient d'autres éléments boucles, alors la succession des variables figurant dans ces éléments est conforme à l'ordre d'exécution des boucles emboîtées (voir p. 8.6.7).

EXEMPLE 8.26. Considérons un élément boucle donné par la structure || ((X (I + 1, J), Y (I, J + 1), J = 1, 5, 2), A (I), I = 2, 9, 3)||.

Dans ce cas il faut transmettre les valeurs des éléments suivants des tableaux X, Y et A :

X (3, 1), Y (2, 2), X (3, 3), Y (2, 4), X (3, 5), Y (2, 6), A (2),
X (6, 1), Y (5, 2), X (6, 3), Y (5, 4), X (6, 5), Y (5, 6), A (5),
X (9, 1), Y (8, 2), X (9, 3), Y (8, 4), X (9, 5), Y (8, 6), A (8).

8.8.2. Instructions d'entrée-sortie sans format. Ces instructions sont : une instruction de rebobinage, une instruction de lecture accès séquentiel sans format, une instruction d'écriture accès séquentiel

sans format, une instruction de retour, une instruction de fin de fichier, une instruction de lecture accès direct sans format, une instruction d'écriture accès direct sans format, une instruction de recherche d'élément de fichier. Examinons ces instructions. Ici et plus bas on entend par instruction d'entrée-sortie accès séquentiel (direct) une instruction qui transmet des données groupées en fichiers à accès séquentiel (direct) (voir p. 8.7.1).

Une *instruction de rebobinage* a la forme

|| REWIND *a* ||,

où *a* est un numéro de fichier. Cette instruction met le fichier de numéro *a* à l'état initial, i.e. prépare à la lecture ou à l'écriture le premier enregistrement de ce fichier.

Une *instruction de lecture accès séquentiel sans format* a la forme

|| READ (*a*) *c* ||,

où *a* est un numéro de fichier, *c* une liste d'entrée (voir p. 8.8.1). Cette instruction réalise l'introduction de l'enregistrement courant du fichier numéro *a* dans la mémoire principale selon la liste *c*. On suppose que les données de fichiers sont représentées en code machine interne.

Si la liste d'entrée n'est pas présente dans l'instruction considérée, l'action de celle-ci se réduit au passage d'un enregistrement courant du fichier à son enregistrement suivant.

Exemple 8.27. La notation || READ (5) A || est une instruction de lecture accès séquentiel sans format.

Elle réalise la transmission de l'enregistrement courant d'un mot en code machine interne du fichier numéro 5 dans la mémoire principale déterminée par la variable A.

Une *instruction d'écriture accès séquentiel sans format* a la forme

|| WRITE (*a*) *c* ||,

où *a* est un numéro de fichier, *c* une liste de sortie (voir p. 8.8.1). Cette instruction prélève sur la mémoire principale les données, représentées en code machine interne, et crée l'enregistrement (sans modifier la forme des données) de la séquence de valeurs (enregistrement courant du fichier *a*), spécifiées par la liste *c*.

Exemple 8.28. La notation || WRITE (6) X || est une instruction d'écriture accès séquentiel sans format qui recopie les données stockées dans la mémoire principale déterminée par la variable X, en les représentant comme enregistrement courant du fichier 6.

Une *instruction de retour* a la forme

|| BACKS PACE *a* ||,

où *a* est un numéro de fichier. Cette instruction agit de telle sorte que ce qui constituait l'enregistrement courant du fichier numéro *a* (avant l'exécution de cette instruction) devient l'enregistrement précédent. Si l'enregistrement courant est le premier enregistrement du fichier, l'instruction de retour ne réalise aucune action.

Une *instruction de fin de fichier* a la forme

|| END FILE *a* ||,

où *a* est un numéro de fichier. Cette instruction transforme l'enregistrement courant du fichier numéro *a* en l'enregistrement final.

Une *instruction de lecture accès direct sans format* a la forme

|| READ (*a'n*) *c* ||,

où *a* est un numéro de fichier, *n* un numéro d'enregistrement (un entier non signé) de ce fichier, *c* une liste d'entrée (voir p. 8.8.1). L'exécution de cette instruction se réduit à l'introduction du *n*-ème enregistrement (en code machine) du fichier numéro *a* dans la zone de mémoire principale déterminée par la liste *c*.

Une *instruction d'écriture accès direct sans format* a la forme

|| WRITE (*a'n*) *c* ||,

où *a* est un numéro de fichier, *n* un numéro d'enregistrement du fichier et *c* une liste de sortie (voir p. 8.8.1).

A la suite de l'exécution de cette instruction, l'information stockée (en code machine) dans la zone de mémoire principale selon la liste *c* est représentée comme *n*-ème enregistrement du fichier *a*.

Une *instruction de recherche d'enregistrement de fichier* a la forme

|| FIND (*a'n*) ||,

où *a* est un numéro de fichier, *n* un numéro d'enregistrement de ce fichier.

Cette instruction positionne l'enregistrement numéro *n* du fichier *a* de telle sorte que la transmission de cet enregistrement demande le minimum du temps.

Une instruction de recherche s'exécute en même temps que les instructions qui la suivent dans un programme FORTRAN. Son exécution se termine lorsque le *n*-ème enregistrement du fichier est trouvé et mis à l'état demandé, ou bien lorsque l'exécution d'une autre instruction d'entrée-sortie est commencée.

8.8.3. Instruction d'entrée-sortie avec format. Ce sont : une instruction de lecture accès séquentiel avec format, une instruction d'écriture accès séquentiel avec format, une instruction de lecture accès direct avec format, une instruction d'écriture accès direct avec format. Considérons ces instructions.

Une *instruction de lecture accès séquentiel avec format* a la forme

|| READ (*a*, *b*) *c* ||,

où *a* est un numéro de fichier, *b* une étiquette d'instruction format (voir p. 8.8.4), *c* une liste d'entrée (voir p. 8.8.1). Cette instruction réalise l'entrée successive des éléments du format dans la zone de mémoire principale déterminée par la liste *c*, avec conversion des données en code machine. La quantité des données à introduire et leur structure sont précisées dans l'instruction FORMAT étiquetée *b*.

Une *instruction d'écriture accès séquentiel avec format* a la forme

|| WRITE (*a*, *b*) *c* ||,

où *a* est un numéro de fichier, *b* une étiquette d'instruction FORMAT (voir. p. 8.8.4), *c* une liste de sortie (voir p. 8.8.1). Cette instruction réalise la sortie successive de l'information de la zone de mémoire principale déterminée par la liste *c*.

Une *instruction de lecture accès direct avec format* a la forme

|| READ (*a'n*, *b*) *c* ||,

où *a* est un numéro de fichier, *n* un numéro d'enregistrement, *b* une étiquette d'instruction FORMAT (voir p. 8.8.4), *c* une liste d'entrée (voir p. 8.8.1).

Cette instruction réalise la lecture des données d'un ou de plusieurs enregistrements du fichier *a*, à partir de l'enregistrement numéro *n*, leur mise en forme machine et leur implantation dans les zones de mémoire principale déterminées par la liste d'entrée *c*. L'information sur la quantité des enregistrements à lire et sur la forme de représentation des données dans les enregistrements est contenue dans l'instruction FORMAT étiquetée *b*.

Une *instruction d'écriture accès direct avec format* a la forme

|| WRITE (*a'n*, *b*) *c* ||,

où *a* est un numéro de fichier, *n* un numéro d'enregistrement, *b* une étiquette d'instruction FORMAT (voir p. 8.8.4), *c* une liste de sortie (voir p. 8.8.1).

Cette instruction extrait les données de la mémoire principale conformément à la liste de sortie *c*, les convertit et crée un ou plusieurs enregistrements du fichier *a*, à partir de l'enregistrement numéro *n*. L'information sur la quantité des enregistrements et sur la forme de représentation des données est contenue dans l'instruction FORMAT étiquetée *b*.

8.8.4. Instruction FORMAT. En examinant les instructions d'entrée-sortie avec format, nous avons dit que l'exécution d'une telle instruction est commandée par une instruction FORMAT.

Celle-ci a la forme

$$\|\text{ FORMAT } (a)\|,$$

où a est une expression de format. Dans le cas le plus simple elle représente une suite $a_1, a_2, \ldots, a_n$ de spécificateurs de types de format sans constantes de répétition (voir § 8.9).

Pour abréger l'écriture d'une expression de format on admet les conventions suivantes :

1. Si une expression de format contient une sous-suite $a_i, a_i, \ldots$ $\ldots, a_i$ de m éléments dont chacun représente un même spécificateur a_i, c'est-à-dire que l'expression de format s'écrit $a_1, \ldots, a_{i-1}$, $a_i, a_i, \ldots, a_i, a_{i+1}, \ldots, a_n$, alors la sous-suite en question peut être remplacée dans l'expression de format par une notation équivalente ma_i, où m est un entier non signé appelé *constante* (*facteur*) *de répétition d'un spécificateur*. Ainsi, l'expression de format mentionnée peut être notée d'une façon plus compacte :

$$a_1, \ldots, a_{i-1}, ma_i, a_{i+1}, \ldots, a_n.$$

2. Si une expression de format contient une sous-suite de spécificateurs $a_i, a_{i+1}, \ldots, a_{i+m}, a_i, a_{i+1}, \ldots, a_{i+m}, \ldots, a_i$, $a_{i+1}, \ldots, a_{i+m}$, qui n'est autre chose qu'une séquence de spécificateurs $a_i, a_{i+1}, \ldots, a_{i+m}$ répétée k fois, alors cette première sous-suite peut être remplacée dans l'expression de format par la notation $k\ (a_i, a_{i+1}, \ldots, a_{i+m})$, où k est un entier non signé qu'on appelle *constante de répétition d'un groupe de spécificateurs*, la notation (a_i, a_{i+1}, a_{i+m}) étant un *groupe de spécificateurs de niveau un*. Si, à son tour, un groupe de niveau un contient une suite de spécificateurs qui représente une sous-suite de spécificateurs répétée plusieurs fois, on y applique la même méthode d'abréger la notation. La notation ainsi obtenue s'appelle *groupe de spécificateurs de niveau deux*. Il n'y a pas dans le FORTRAN de groupe de niveau supérieur à deux.

EXEMPLE 8.29. Considérons une structure $\|$ FORMAT (a_1, a_2, a_3, a_4, a_5, a_3, a_4, a_5, a_3, a_4, a_5, a_6, a_7, a_8, a_7, a_8, a_7, a_8, a_7, a_8, a_9)$\|$ qui est une instruction FORMAT, a_i étant des spécificateurs de type.

Cette instruction est équivalente à la notation $\|$ FORMAT (a_1, a_2, 3 (a_2, a_4, a_5), a_6, 4 (a_7, a_8), a_9)$\|$.

EXEMPLE 8.30. Considérons une structure $\|$ 4 FORMAT (a_1, a_2, a_3, a_4, a_5, a_5, a_5, a_3, a_4, a_5, a_5, a_5, a_2, a_3, a_4, a_5, a_5, a_5, a_3, a_4, a_5, a_5, a_5)$\|$ qui est une instruction FORMAT, a_i étant des spécificateurs de type.

Ecrivons cette instruction sous une forme condensée, équivalente $\|$ 4 FORMAT (a_1, 2 (a_2, 2 (a_3, a_4, a_5, a_5, a_5,)))$\|$. Ici (a_2, 2 (a_3, a_4, a_5, a_5, a_5)) est un groupe de spécificateurs de niveau deux, (a_3, a_4, a_5, a_5, a_5) un groupe de niveau un.

Mais une instruction de la forme || 4 FORMAT (a_1, 2 (a_3, 2 (a_3, a_4, 3 (a_5)))) || est inadmissible dans le FORTRAN.

Des instructions FORMAT sont utilisées par les instructions d'entrée-sortie. Nous allons formuler quelques règles d'utilisation correcte des instructions FORMAT dans le cas d'une expression de format simple.

Tous les spécificateurs de type (voir p. 8.9.1) sont divisés en deux groupes selon leur destination dans le FORTRAN : le premier groupe comprend les spécificateurs servant à décrire la structure des valeurs des quantités à transmettre (c'est le cas des spécificateurs I, F, E, D, G, L, A, Z) ; le deuxième groupe réunit les spécificateurs servant à commander l'entrée-sortie et utilisés à l'édition (c'est le cas des spécificateurs X, T).

Il est établi une correspondance entre les spécificateurs du premier groupe de l'expression de format d'une instruction FORMAT (pour fixer les idées, nous convenons que cette expression contient k spécificateurs) et les variables de la liste d'une instruction d'entrée-sortie liée à l'instruction FORMAT (convenons que la liste comprenne m variables). Cette correspondance est définie par les règles :

1. Si $k = m$, alors à la première variable de la liste il correspond le premier spécificateur du premier groupe, en suivant l'expression de la gauche vers la droite ; à la deuxième variable de la liste il correspond le deuxième spécificateur du premier groupe de l'expression de format, etc., à la m-ème variable de la liste il correspond le k-ème spécificateur dans l'expression de format.

Dans ce cas, une sortie commence par essayer les spécificateurs de l'instruction FORMAT de la gauche vers la droite, et, si un spécificateur essayé appartient au premier groupe, on prend dans la liste de l'instruction de sortie la variable correspondante. Sa valeur est mise sous la forme déterminée par le spécificateur et est placée dans le champ de l'enregistrement. Si le spécificateur essayé appartient au deuxième groupe, alors les actions nécessaires sont accomplies sans consulter la liste de sortie.

Le cas d'une instruction d'entrée est tout à fait analogue.

2. Si $k < m$, alors aux k premiers éléments de la liste il correspond tous les k spécificateurs du premier groupe dans l'expression de format, au $(k + 1)$-ème élément de la liste, le premier spécificateur du premier groupe de l'expression en comptant de la gauche vers la droite, au $(k + 2)$-ème élément, le deuxième spécificateur du premier groupe de l'expression, etc. Le présent cas se réduit donc à des applications itérées de la règle précédente, pour chaque collection de k éléments successifs de la liste, à la seule différence près que chaque nouvel examen des spécificateurs de l'expression de format commence par un passage à l'enregistrement suivant du fichier.

Les entrées-sorties se réalisent dans ce cas exactement de même que dans le cas précédent.

3. Si $k > m$, alors aux variables de la liste d'entrée (sortie) il correspond les m premiers spécificateurs du premier groupe de l'expression de format. La correspondance est établie suivant la règle du point 1.

Les entrées-sorties se réalisent comme dans le cas décrit au point 1.

Exemple 8.31. Soit un fragment de programme FORTRAN:

|| 102 FORMAT (a_1, a_2, a_3)
A = 2.0 + A
READ (b, 102) A, B1, X, Y||,

où A, B1, X et Y sont des variables.

En exécutant l'instruction d'entrée

|| READ (b, 102) A, B1, X, Y||

qui est liée à l'instruction

|| 102 FORMAT (a_1, a_2, a_3)||

on affectera aux variables A, B1, X et Y les valeurs coïncidant avec les données du fichier b dont la structure est déterminée par les spécificateurs respectifs a_1, a_2, a_3 et a_1.

Si la liste d'entrée (sortie) contient un tableau, alors, au cours de l'interaction d'instruction d'entrée-sortie avec l'instruction FORMAT, ce tableau est considéré comme une suite dont chaque élément représente une variable indicée. Donc tout ce qu'on a dit plus haut sur l'interaction des instructions d'entrée-sortie avec une instruction FORMAT reste en vigueur.

Exemple 8.32. Supposons qu'un programme FORTRAN contienne les instructions

|| 6 FORMAT (a_1, a_2)||
WRITE (c, 6) A, B||

où A est une variable, B un tableau unidimensionnel se composant de quatre éléments.

A la suite de l'exécution de l'instruction || WRITE (c, 6) A, B|| liée à l'instruction || 6 FORMAT (a_1, a_2)|| il sera créé un enregistrement courant d'un fichier, qui n'est pas celui d'une imprimante, à partir des valeurs de la variable A et des éléments B (1), B (2), B (3), B (4) du tableau B, la structure de ces valeurs étant déterminée par les spécificateurs respectifs a_1, a_2, a_1, a_2, a_1.

On a parfois besoin de sauter un nombre d'enregistrements d'un fichier au cours d'une entrée ou d'une sortie. Cette possibilité est prévue dans le FORTRAN, et l'information correspondante est donnée dans l'instruction FORMAT sous forme de symboles || / || dont le nombre est égal à celui d'enregistrements sautés.

Exemple 8.33. Considérons l'instruction

|| 4 FORMAT (// a_1 /// a_2);

d'un programme FORTRAN.

Si cette instruction est liée à une instruction d'entrée, alors l'exécution de cette dernière commence par sauter le premier et le deuxième enregistrement du fichier donné, ensuite sont lus et introduits dans la mémoire principale les éléments du troisième enregistrement du fichier décrits par le spécificateur a_1, puis on saute encore trois enregistrements, et enfin on réalise l'entrée des élements du septième enregistrement décrits par le spécificateur a_2.

Dans le cas général, l'interaction d'une instruction d'entrée-sortie avec une instruction FORMAT se réalise de la manière suivante.

Une sortie commence par examiner successivement, de la gauche vers la droite, les spécificateurs (compte tenu des constantes de répétition et des parenthèses). Lorsqu'un spécificateur examiné appartient au premier groupe, on choisit dans la liste de sortie la variable correspondante. La valeur de celle-ci est mise sous la forme qui correspond au spécificateur choisi et est transférée dans le champ d'enregistrement. Lorsque le spécificateur examiné appartient au deuxième groupe, les actions nécessaires s'exécutent sans utiliser la liste de sortie.

Si la liste de sortie est épuisée avant que le soit l'expression de format, alors l'examen des spécificateurs se poursuit jusqu'à satisfaire à l'une des conditions suivantes:

a) un spécificateur du premier groupe se rencontre au cours des examens;

b) l'expression de format est épuisée.

Ceci fait, l'exécution de l'instruction de sortie se termine.

Dans le cas où l'expression de format s'épuise avant la liste de sortie, on répète les essais de spécificateurs, à commencer par le premier spécificateur du dernier groupe de niveau un. Une itération d'examens d'une expression de format ne contenant pas de groupes commence par le premier spécificateur.

Chaque nouveau passage sur l'expression de format est accompagné par un passage à l'enregistrement suivant du fichier. On change d'enregistrement également chaque fois qu'on tombe, au cours de l'examen des spécificateurs, sur un symbole || / ||.

Si l'enregistrement courant n'est pas utilisé, et qu' il faille passer à l'enregistrement suivant, alors l'enregistrement courant est sauté.

L'interaction d'une instruction d'entrée avec une instruction FORMAT se réalise de manière analogue.

§ 8.9. Descriptions dans les programmes FORTRAN

8.9.1. Spécificateurs de type. Un spécificateur (descripteur) de type (code de format) sert à décrire le type et la structure des données d'un fichier. Il détermine également la quantité des données à faire entrer (sortir) d'un fichier d'un type donné. Il a été dit qu'on distingue deux espèces de spécificateurs de type. Les spécificateurs I, F, E, D, G, L, A, Z sont de première espèce. Ils servent à décrire le type des valeurs de variables. Les spécificateurs des types « format symbolique », X, T sont de deuxième espèce. On s'en sert pour l'édition. Nous allons examiner chaque spécificateur de plus près.

Le *spécificateur de type I* sert à décrire la forme de représentation des valeurs de variables entières, il s'écrit

$$\| m\mathrm{I}w \|$$

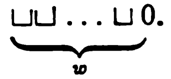
où $\| \mathrm{I}w \|$ est un spécificateur de type proprement dit, w, le champ total, m, la constante de répétition du spécificateur (voir p. 8.8.4) (w et m sont des entiers sans signe). Dans la liste d'une instruction d'entrée-sortie liée à l'instruction FORMAT en question, il correspond à ce spécificateur de type m variables entières. Si $m = 1$. on peut omettre ce nombre dans la notation du spécificateur.

Si la valeur d'une variable est un nombre entier non nul, ce nombre sera représenté dans le champ d'enregistrement, conformément au spécificateur du type I, comme

$$\underbrace{\sqcup\sqcup \ldots \sqcup \omega a_1, a_2, \ldots, a_k}_{w},$$

où $\sqcup$ est le symbole d'espace; ω est soit le symbole d'espace (si le nombre entier est positif) soit le signe « moins » (si ce nombre est négatif); a_i ($i = 1, 2, \ldots, k$) sont des chiffres ($a_1 \neq 0$).

Si la valeur de la variable est un entier nul, elle sera représentée



$$\underbrace{\sqcup\sqcup \ldots \sqcup 0}_{w}.$$

Si le champ total est plus petit que le nombre de symboles nécessaire à la représentation de la variable, alors, en sortie, toutes les positions du champ seront remplies d'astérisque $\| * \|$.

Exemple 8.34. Supposons que les variables entières I, J, K, L, M aient pour valeurs les nombres 375, —516, 00124, 0, 246000123 respectivement. Dans ces conditions, à la suite d'exécution de l'instruction de sortie || WRITE (4'2, 3) I, J, K, L, M|| liée à l'instruction FORMAT || 3 FORMAT (517)||, les valeurs en question seront écrites dans le deuxième enregistrement du ficher 4 comme

|| ⊔ ⊔ ⊔ ⊔ 375 || ⊔ ⊔ ⊔ —516 || ⊔ ⊔ 00124 ||
⊔ ⊔ ⊔ ⊔ ⊔ ⊔ 0 || ******* ||

respectivement.

Exemple 8.35. Supposons que le sixième enregistrement du fichier numéro 3 ait la forme

|| ⊔ ⊔ ⊔ ⊔ ⊔ 27 ⊔ ⊔ ⊔ —318 ⊔ ⊔ ⊔ ⊔ 700 ⊔ ⊔ ⊔ ⊔ ⊔ ⊔
5 ⊔ ⊔ ⊔ ⊔ ⊔ —3||.

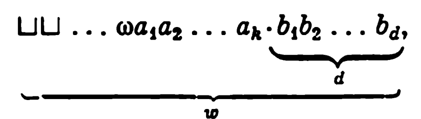
Dans ces conditions, à la suite d'exécution de l'instruction d'entrée || READ (3'6,10) A, I, K, L|| liée à l'instruction FORMAT || FORMAT (417)||, les variables entières A, I, K, L auront pour valeurs les nombres entiers respectifs 27, —318, 700, 5 représentés en code machine.

Le *spécificateur de type F* sert à décrire la forme de représentation des valeurs de variables réelles, si ces valeurs sont des nombres réels sans exposant. Il s'écrit

|| mF$w.d$||,

où || F$w.d$|| est le spécificateur de type proprement dit, w le champ total d'un enregistrement, m une constante de répétition, d la longueur de la partie fractionnaire d'un nombre réel sans exposant (w, m et d sont des entiers non signés). Dans la liste d'une instruction d'entrée-sortie liée à l'instruction FORMAT en question, il correspond au spécificateur de type m variables réelles. Si $m = 1$, on peut omettre ce nombre dans la notation du spécificateur.

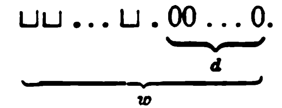
Si une variable réelle a pour valeur un nombre réel sans exposant et avec partie entière non nulle, ce nombre sera représenté, conformément à un spécificateur de type F, comme



$$\underbrace{\sqcup\sqcup \ldots \omega a_1 a_2 \ldots a_k \cdot \underbrace{b_1 b_2 \ldots b_d}_{d}}_{w},$$

où || ⊔ || est le symbole d'espace; ω est soit un espace (si le nombre est positif), soit un signe « moins » (si le nombre est négatif); a_i ($i = 1, 2, \ldots, k$) sont des chiffres de la partie entière du nombre ($a_1 \neq 0$), b_j ($j = 1, 2, \ldots, d$) ceux de la partie fractionnaire.

Si la valeur de la variable est un nombre réel nul sans exposant, et que $d \neq 0$, alors ce nombre sera représenté comme



Si la valeur de la variable est un nombre réel nul sans exposant, et que $d = 0$, alors il sera représenté comme || ⊔ ⊔ . . . ⊔ 0.||. Enfin, si le nombre total des symboles formant la partie entière d'un nombre réel sans exposant (y compris le signe et le point décimal) dépasse $w - d$, alors ce nombre sera représenté comme ||$\underbrace{** \ldots *}_{w}$||.

Exemple 8.36. Supposons que les variables réelles sans exposant A, B, C, X et Y aient pour valeurs respectivement les nombres —3.07, 20.01, 1., —16270043.02, 0.

Dans ces conditions, à la suite d'exécution de l'instruction ||WRITE (8'3,2) A, B, C, X, Y|| liée à l'instruction FORMAT ||2 FORMAT (5F9.3)||, les valeurs indiquées seront représentées dans le troisième enregistrement du fichier numéro 8 comme

|| ⊔ ⊔ ⊔ — 3.070|| ⊔ ⊔ ⊔ 20.010|| ⊔ ⊔ ⊔ ⊔ 1.000|| *********||
1|| ⊔ ⊔ ⊔ ⊔ ⊔. 000||

respectivement.

Exemple 8.37. Supposons que le quatrième enregistrement du fichier numéro 12 ait la forme

|| ⊔ ⊔ —6,071860.00 ⊔ —34.21 — 600.08 ⊔ ⊔ ⊔ ⊔.29||.

A la suite d'exécution de l'instruction d'entrée || READ (12'4,3) X, Y, Z, S|| liée à l'instruction FORMAT|| 3 FORMAT (4F7.2)||, les variables réelles X, Y, Z et S auront pour valeurs —6.07, 1860.00, —34.21 et —600.08 respectivement.

Le *spécificateur de type E* sert à décrire la forme de représentation des valeurs des variables réelles, si ces valeurs sont des nombres réels simple précision avec exposant ou des composantes de variables complexes simple précision avec exposant. Il s'écrit

|| mE$w.d$||,

où ||E$w.d$|| est le spécificateur de type proprement dit; w, le champ total; m la constante de répétition du spécificateur (voir p. 8.8.4); d, la longueur de partie fractionnaire de la mantisse (w, m et d

sont des entiers non signés). Si $m = 1$, on peut omettre ce nombre dans le spécificateur. La condition $w \gg d + 7$ est à estimer.

Si la valeur d'une variable est un nombre réel simple précision avec exposant, ce nombre sera représenté, à l'aide du spécificateur de type E, comme

$$\underbrace{\sqcup\sqcup \ldots \sqcup \omega_1 0 \cdot b_1 b_2 \ldots b_d \mathrm{E} \omega_2 a_1 a_2}_{w},$$

où $\| \sqcup \|$ est le symbole d'espace; ω_1 le symbole $\| \sqcup \|$ si le nombre est positif ou le signe « moins » s'il est négatif; b_i $(i = 1, 2, \ldots, d)$ sont les chiffres de la partie fractionnaire de la mantisse ($b_1 \neq 0$ si le nombre n'est pas nul, $b_1 = 0$ s'il est nul); a_j $(j = 1, 2)$ sont les chiffres de l'exposant; ω_2 est le symbole $\| \sqcup \|$ si l'exposant est un nombre positif ou nul ou le signe « moins » si l'exposant est un nombre négatif.

Si la longueur de la partie fractionnaire de la mantisse est supérieure à $w - 4$, le nombre est arrondi selon les règles usuelles.

EXEMPLE 8.38. Supposons que les variables réelles X, Y, Z, A, S aient pour valeurs respectivement les nombres —304.2, 0.00000000000061, 0.0, —0.00003, —76537400.0.

A la suite d'exécution de l'instruction de sortie ||WRITE (6'1,4) X, Y, Z, A, S|| liée à l'instruction FORMAT|| 4 FORMAT (3E12.3, 2E10.2)||, les valeurs des variables indiquées seront représentées dans le premier enregistrement du fichier numéro 6 comme || ⊔ ⊔ —0.3042EO3|| ⊔ ⊔ ⊔ 0.610E — 12|| ⊔ ⊔ ⊔ 0.000E ⊔00|| ⊔ —0.30E — 04|| ⊔ —0.77E ⊔ 08|| respectivement.

Le *spécificateur de type D* sert à décrire la forme de représentation des valeurs des variables réelles, lorsque ces valeurs sont des nombres réels double précision avec exposant ou des composantes de variables complexes double précision. Il s'écrit

$$\| m\mathrm{D}w.d \|,$$

où $\| \mathrm{D}w.d \|$ est le spécificateur de type proprement dit; w, m et d ont le même sens que pour le spécificateur de type E et déterminent la structure du nombre de même que le spécificateur de type E, avec la seule différence que la lettre E dans la notation de l'exposant est remplacée par la lettre D.

Le *spécificateur de type L* sert à décrire la forme de représentation des valeurs des variables logiques. Il s'écrit

$$\| m\mathrm{L}w \|,$$

où $\| \mathrm{L}w \|$ est le spécificateur de type proprement dit, w le champ total, m la constante de répétition du spécificateur (m, w sont

des entiers non signés). Si $m = 0$, on peut omettre ce nombre dans le spécificateur.

La valeur d'une variable logique, conformément au spécificateur de type L, aura la forme

$$\underbrace{\sqcup\sqcup \ldots \sqcup}_{w} a,$$

où || ⊔ || est le symbole d'espace, a le symbole || T|| si la valeur de variable logique est || .TRUE.|| ou le symbole || F|| si cette valeur est || .FALSE.||.

Le *spécificateur de type « format symbolique »* sert à décrire une ligne et a la forme

$$a,$$

où a est une ligne avec ou sans indicateur de longueur (voir § 8.2). Le champ total sera déterminé, dans le deuxième cas, par le nombre des symboles constituant le corps de la ligne.

Au cours d'une sortie, le corps de la ligne contenue dans le spécificateur « format symbolique » de l'instruction FORMAT liée à l'instruction de sortie considérée, est placé dans le champ d'enregistrement. S'il s'agit d'une entrée, la suite de symboles contenue dans le champ d'enregistrement remplace le corps de ligne du spécificateur « format symbolique » de l'instruction FORMAT.

EXEMPLE 8.39. A la suite d'interaction de l'instruction de sortie ||WRITE (5'7,3)|| avec l'instruction FORMAT ||3 FORMAT ('X = = 0.3425⊔⊔')||, la suite de caractères || X = 0.3425⊔⊔ || sera stockée dans le champ du septième enregistrement du fichier numéro 5.

EXEMPLE 8.40. Supposons que le onzième enregistrement du fichier 6 ait la forme

|| IVANOV FOR X = 0.34⊔⊔—6.1 ||.

A la suite d'exécution de l'instruction d'entrée ||READ (6'11,4)|| liée à l'instruction FORMAT|| 4 FORMAT ('X=—160745.03⊔⊔⊔)|| le spécificateur || 'X = —160745.03⊔⊔⊔' || de l'instruction FORMAT donnée sera remplacé par le spécificateur || 'IVANOV FOR X = 0.34' ||. Ainsi, l'instruction FORMAT donnée changera en || 4FORMAT ('IVANOV FOR X = 0.34') ||.

Le *spécificateur de type X* a la forme

|| wX ||.

C'est un descripteur de zones de blancs.

Son rôle est déterminé par l'instruction d'entrée-sortie liée à l'instruction FORMAT qui contient le spécificateur en question. En entrée, il fait que w caractères de l'enregistrement externe sont sautés (ne sont pas introduits dans la mémoire principale); en sortie, il détermine l'insertion de w symboles || ⊔ || (blancs) dans l'enregistrement externe.

Le *spécificateur de type A* est un descripteur des chaînes de caractères alphanumériques. Il a la forme

$$\| mAw \|,$$

où $\| Aw \|$ est le spécificateur de type proprement dit, m, la constante de répétition du spécificateur, w, le champ total de l'enregistrement (m et w sont des entiers non signés). Si $m = 1$, on peut omettre ce nombre dans le spécificateur.

Dans une liste d'entrée-sortie, il doit correspondre à ce spécificateur des variables entières ou réelles, ainsi que des composantes de variables complexes. Ces grandeurs peuvent prendre des valeurs symboliques. Le nombre l des symboles dans une valeur de variable est fixé comme suit : la longueur l vaut 2 octets pour une variable entière longueur non standard, 4 octets pour une variable entière, réelle ou complexe longueur standard, 8 octets pour une variable réelle ou complexe longueur non standard.

Règles d'inscription dans les champs d'enregistrement en entrée (sortie)

Le remplissement des champs d'enregistrement par des valeurs de variables au cours d'une entrée-sortie se fait selon les règles suivantes :

1. Soit $w = l$. Alors : a) si un champ d'enregistrement est rempli par les symboles $s_1 s_2 \ldots s_w$, la variable correspondante recevra en entrée la valeur $s_1 s_2 \ldots s_w$; b) si une variable a la valeur symbolique $s_1 s_2 \ldots s_l$, le champ correspondant d'enregistrement sera rempli en sortie par les symboles $s_1 s_2 \ldots s_l$.

2. Soit $w > l$. Alors : a) si un champ d'enregistrement est rempli par les symboles $s_1 s_2 \ldots s_w$, la variable correspondante recevra en entrée la valeur symbolique $s_{w-l+1} s_{w-l+2} \ldots s_w$; b) si une variable a la valeur symbolique $s_1 s_2 \ldots s_l$, le champ correspondant d'enregistrement sera rempli en sortie par les symboles ⊔⊔ ... ⊔$s_1 s_2 \ldots s_l$, où || ⊔⊔ ... ⊔ || sont des symboles d'espace (leur nombre est $w - l$).

3. Soit $w < l$. Alors : a) si un champ d'enregistrement est rempli par les symboles $s_1 s_2 \ldots s_w$, la variable correspondante recevra en entrée la valeur symbolique $s_1 s_2 \ldots s_w$ ⊔⊔ ... ⊔, où || ⊔⊔ ... ⊔ || sont des symboles d'espace (leur nombre est $l - w$) ; b) si une variable a la valeur symbolique $s_1 s_2 \ldots s_l$, le champ correspondant d'enregistrement sera rempli en sortie par les symboles $s_1 s_2 \ldots s_w$.

EXEMPLE 8.41. Supposons que le septième enregistrement du fichier 10 ait la forme

|| 6A7B327034—001602800ALFANOF1234567 ||.

A la suite d'exécution de l'instruction || READ (10'7,8) I, J, X, A|| liée à l'instruction FORMAT || 8 FORMAT (4A4)||, la variable entière I longueur non standard, la variable entière J longueur standard, la variable réelle X simple précision et la variable réelle A double précision prendront les valeurs respectives: || 7B|| 3270|| 34—0|| 0160⊔⊔⊔⊔||. Soit l'instruction|| WRITE (10'7,14) K, B, C|| liée à l'instruction || 14 FORMAT (3A5)||, où K est une variable entière longueur standard ayant pour valeur la suite de symboles || A21N||, B est une variable réelle double précision ayant pour valeur la suite || ⊔IVANOV⊔ ||, et C est une variable réelle simple précision ayant pour valeur la suite || A/34||. Alors le septième enregistrement du fichier 10 sera modifié comme suit: || ⊔A21N⊔ IVAN⊔ A/3402800 ALFANOF 1234567||.

Le *spécificateur de type Z* sert à décrire les nombres hexadécimaux *). Il a la forme

|| mZw ||,

où || Zw|| est le spécificateur de type proprement dit; w, le champ total; m, la constante de répétition du spécificateur (w et m sont des entiers non signés). Si $m = 1$, on peut omettre ce nombre dans le spécificateur.

La liste d'une instruction d'entrée-sortie liée à une instruction FORMAT contenant le spécificateur en question doit contenir m grandeurs. Celles-ci peuvent être des variables entières ou réelles ou des composantes de variables complexes. Les valeurs de ces variables sont des suites de chiffres hexadécimaux. Le nombre l de chiffres dans la valeur d'une variable est déterminé de la manière suivante: $l = 4$ pour une variable entière longueur non standard; $l = 8$ pour une variable entière longueur standard; $l = 8$ pour une variable réelle et une composante de variable complexe simple précision; $l = 16$ pour une variable réelle ou une composante de variable complexe double précision.

Le remplissement des champs d'enregistrement selon le spécificateur de type Z se fait en estimant les règles suivantes:

1. Soit $w = l$. Dans ce cas: a) si un champ d'enregistrement est rempli par les chiffres $s_1s_2 \ldots s_w$, alors, en entrée, la variable correspondante prendra la valeur $s_1s_2 \ldots s_w$; b) si une variable a pour valeur $s_1s_2 \ldots s_w$, alors, en sortie, le champ correspondant d'enregistrement sera rempli par les chiffres $s_1s_2 \ldots s_l$.

*) En tant que chiffres hexadécimaux, on utilise dans le FORTRAN les symboles || 0|| 1|| 2|| 3 || 4|| 5|| 6|| 7|| 8|| 9|| A|| B|| C|| D|| E|| F||.

2. Soit $w > l$. Dans ce cas: a) si un champ d'enregistrement est rempli par les chiffres $s_1s_2 \ldots s_w$, alors, en entrée, la variable correspondante prendra la valeur $s_{w-l+1}s_{w-l+2} \ldots s_w$; b) si une variable a pour valeur $s_1s_2 \ldots s_l$, alors, en sortie le champ d'enregistrement correspondant sera rempli par les symboles

$$\sqcup\sqcup \ldots \sqcup s_1s_2 \ldots s_l,$$

où || ⊔ || est le symbole d'espace (le nombre de blancs est $w - l$).

3. Soit $w < l$. Dans ce cas: a) si un champ d'enregistrement est rempli par les chiffres $s_1s_2 \ldots s_w$, alors, en entrée, la variable correspondante prendra la valeur

$$s_1s_2 \ldots s_w\ 00 \ldots 0,$$

où 00 . . . 0 est une suite de $l - w$ zéros hexadécimaux; b) si une variable a $s_1s_2 \ldots s_l$ pour valeur, alors, en sortie, le champ d'enregistrement correspondant sera rempli par les chiffres $s_1s_2 \ldots s_w$.

Exemple 8.42. Supposons que le septième enregistrement du fichier 10 ait la forme

|| 1234A3F210B0F0C2579DA8763210067A ||.

A la suite d'exécution de l'instruction || READ (10'7,8) I, J, X, Y || liée à l'instruction FORMAT || 8 FORMAT (4Z8) ||, la variable entière I longueur standard prendra la valeur || 1234A3F2 ||, la variable entière J longueur non standard la valeur || F0C2 ||, la variable réelle X double précision la valeur || 2579DA8700000000 ||, la variable réelle Y simple précision la valeur || 63210067 ||.

Soit l'instruction || WRITE (10'7,4) K, B, A || liée à l'instruction || 4FORMAT (3Z8) ||, où K est une variable entière longueur non standard ayant la valeur || A230 ||, B, une variable réelle simple précision de valeur || 760089A3 || et A, une variable réelle double précision de valeur || 6012AD340000AF34 ||. Alors le septième enregistrement du fichier 10 sera transformé en

|| ⊔⊔⊔⊔ A230760089A36012AD34 || 3210067A ||.

Le *spécificateur de type G* sert à décrire les valeurs de variables entières, réelles, complexes et logiques. Il a la forme

|| mG$w.d$ ||,

où || G$w.d$ || est le spécificateur de type proprement dit; w, le champ total d'enregistrement; m, la constante de répétition du spécificateur; d, le nombre de chiffres significatifs dans la partie fractionnaire (w, m et d sont des entiers non signés). Si $m = 1$, on peut omettre ce nombre dans le spécificateur de type.

Si un spécificateur G fait partie d'une instruction FORMAT liée à une instruction de sortie, on peut l'utiliser à la place des spécificateurs de types D, E, F, I, L, puisque:

— un spécificateur || G*w*.*d*||, s'il lui correspond dans la liste de sortie une variable entière, est équivalent à un spécificateur de type I qui a la forme || I*w*||;

— un spécificateur || G*w*.*d*||, s'il lui correspond une variable logique, est équivalent à un spécificateur de type L qui a la forme || L*w*||;

— un spécificateur || G*w*.*d*||, s'il lui correspond une variable réelle ou complexe, est équivalent à une combinaison de spécificateurs F et X qui dépend de la valeur de cette variable. Cette dépendance est explicitée dans la table 8.3.

Table 8.3

Forme de représentation en sortie de la variable en fonction de sa valeur N

Valeur N de la variable	Spécificateur de type équivalent au spécificateur || G*w*. *d* ||
$0.1 \leqslant N < 1$	|| F $(w-4)\cdot d$, 4X ||
$1 \leqslant N < 10$	|| F $(w-4)\cdot(d-1)$, 4X ||
$10 \leqslant N < 10^2$	|| F $(w-4)\cdot(d-2)$, 4X ||
. . . .	. . . .
$10^{d-1} \leqslant N < 10^d$	|| F $(w-4)\cdot 0$, 4X ||

Dans les autres cas, l'action d'un spécificateur [G*w*.*d*|| est équivalente à celle d'un spécificateur || E*w*.*d*|| (pour les variables longueur standard).

Le *spécificateur de type T* indique le numéro de la position d'enregistrement par laquelle commence l'entrée-sortie. Il a la forme

|| T*w*||,

où *w* est le numéro de la position d'enregistrement.

En impression, le spécificateur de type T indique non pas le numéro de la position par laquelle l'impression commence, mais celui de la position suivante.

On peut faire précéder un spécificateur de type D, E ou F par un *facteur d'échelle* ayant la forme

nPa,

où a est l'un des spécificateurs de type énumérés; n, un entier (arbitraire) non signé.

En entrée, le facteur d'échelle n'a pas d'effet si *a* est un spécificateur de type E ou D. Si *a* est un spécificateur de type F, la valeur de chaque variable introduite se multiplie par 10^{-n}.

En sortie, la valeur de la variable se multiplie par 10^n. De plus, si *a* est un spécificateur de type E ou D, l'exposant de cette valeur est diminué de *n*. La correction de l'exposant n'a pas lieu si *a* est un spécificateur de type F.

L'effet d'un facteur d'échelle s'étend à tous les spécificateurs de type E, D et F au cours de la même opération d'entrée-sortie jusqu'à ce qu'un autre facteur d'échelle soit rencontré. Il n'agit pas sur un spécificateur de type I. Un facteur d'échelle peut être un nombre positif ou négatif.

8.9.2. Contrôle de l'impression. Une imprimante est considérée comme un fichier à accès séquentiel dans lequel chaque enregistrement représente une ligne d'impression. L'ensemble d'un certain nombre de lignes forme une page (le nombre de positions dans une ligne et le nombre de lignes dans une page sont déterminés par le type de l'imprimante).

Si l'impression se fait par lignes, le premier symbole de l'enregistrement à imprimer est sauté. Il sert à positionner le papier et s'appelle *caractère de contrôle*. Le contrôle se réalise comme suit: si le caractère de contrôle est un blanc, on passe à la ligne suivante; s'il est un zéro, on saute une ligne, s'il est un « plus », l'impression s'effectue sans changer de ligne, s'il est l'unité, on passe à la première ligne de la page suivante.

Le caractère de contrôle est en général donné par un spécificateur de type « format symbolique » dans une instruction FORMAT.

Lorsque les données destinées à l'impression sont enregistrées sur une bande magnétique ou sur n'importe quel autre support d'information, le caractère de contrôle s'écrit de la même façon que tout autre symbole faisant partie des données.

8.9.3. Instructions non exécutables du FORTRAN. Il existe en FORTRAN trois façons d'indiquer le type d'une grandeur (variable, tableau ou fonction): une déclaration de type automatique (le type est prédéfini dans le FORTRAN par une convention), une déclaration de type implicite et une déclaration de type explicite.

Une *déclaration automatique* n'est admissible que pour définir les types entier et réel. Une grandeur est entière si son nom symbolique commence par l'une des lettres || I || J || K || L || M || N ||; elle est réelle dans les autres cas. La déclaration automatique ne s'applique qu'aux grandeurs longueur standard.

EXEMPLE 8.43. Supposons qu'un programme FORTRAN utilise les grandeurs KIM, IL, FOR non décrites par aucune instruction

de déclaration de type, explicite ou implicite, de ce programme. Dans ce cas, conformément à la convention adoptée, les variables KIM et IL sont considérées comme entières longueur standard et la variable FOR1 comme réelle simple précision.

Une *instruction de déclaration de type implicite* a la forme

$$\|\text{IMPLICIT } t_1s_1c_1, t_2s_2c_2, \ldots, t_ks_kc_k\|,$$

où t_i $(i = 1, 2, \ldots, k)$ sont des descripteurs de type (on utilise en tant que tels les symboles || INTEGER || REAL || COMPLEX || LOGICAL ||; s_i $(i = 1, 2, \ldots, k)$ sont des descripteurs de longueur (on utilise en tant que tels les symboles || * 2 || * 4 || * 8 || * 16 ||) ; c_i $(i = = 1, 2, \ldots, k)$ sont des identificateurs de grandeurs.

Les descripteurs ||INTEGER || REAL || COMPLEX || LOGICAL || décrivent respectivement les types entier, réel, complexe et logique.

Le descripteur de longueur || * 4|| indique la longueur standard de grandeurs entières et réelles ; || * 8|| indique la longueur non standard de grandeurs réelles et la longueur standard de grandeurs complexes; || * 2|| indique la longueur non standard de grandeurs entières; || * 16|| indique la longueur non standard de grandeurs complexes. Pour indiquer la longuer standard de grandeurs de tous les types on utilise le descripteur « vide ».

Une *liste d'identificateurs de grandeurs* a la forme

$$(a_1, a_2, \ldots, a_n),$$

où a_i est soit une lettre par laquelle commence le nom d'une grandeur, soit un identificateur d'une suite de lettres, ayant la forme $b_1 - b_2$, où b_1 et b_2 sont la première et la dernière lettre de la suite; toutes les lettres d'une suite sont disposées dans l'ordre alphabétique et représentent les premières lettres des noms de certaines grandeurs.

Une instruction de déclaration de type implicite permet de décrire le type de chaque grandeur suivant le même principe qu'une déclaration automatique, à ceci près que les lettres initiales des noms de grandeurs sont choisies au gré du programmeur. Il est possible de spécifier tous les types de variables. De plus, une instruction implicite informe le type et la longueur déclarés automatiquement (elle a priorité sur la déclaration automatique).

Exemple 8.44. La notation || IMPLICIT REAL * 8 (M — R, J, A — D), INTEGER * 2 (S, Z, X)|| est une instruction de déclaration de type implicite. Elle déclare que les grandeurs ALPHA, BOK, MIR, RI qui figurent dans le programme sont réelles double précision, tandis que les grandeurs X, Z, XP, SAR sont entières longueur non standard.

Une *instruction de déclaration de type* déclare le type de variables non pas par la première lettre de leur nom, mais d'après tout le nom. Il a la forme $tsa_1s_1\ (k_1)/x_1/,\ a_2s_2\ (k_2)\ /x_2/,\ \ldots,\ a_ns_n\ (k_n)/x_n/$, où t est le descripteur de type ; s le descripteur commun de longueur ; $a_i\ (i = 1, 2, \ldots, n)$ le nom d'une grandeur du type t ; $s_i\ (i = 1, 2, \ldots, n)$ le descripteur de longueur de la grandeur a_i ; $x_i\ (i = 1, 2, \ldots, n)$ la valeur initiale de a_i.

La signification des descripteurs de type et de longueur est la même que pour une instruction de déclaration de type implicite.

Si un descripteur de longueur s_i n'est pas présent, la longueur de la grandeur correspondante est caractérisée par le descripteur commun de longueur s. Si celui-ci n'est pas présent non plus, on considère a_i comme grandeur de longueur standard.

Les paramètres k_i et x_i ne sont pas obligatoires. Le paramètre k_i n'est indiqué que pour les tableaux et a la forme $(T_1, T_2, \ldots, T_m)$ où m est la dimension du tableau, $T_j\ (j = 1, 2, \ldots, m)$ est la valeur maximale de l'indice du tableau pour la j-ème dimension. (Le produit $T_1 \times T_2 \times \ldots \times T_m$ détermine la taille d'un tableau i.e. le nombre de ses éléments.)

Les valeurs initiales ne peuvent être attribuées qu'aux variables et aux tableaux. La valeur initiale d'une variable est donnée par un nombre, celle d'un tableau a la forme d'une liste $c_1, c_2, \ldots, c_p$, où $c_l\ (l = 1, 2, \ldots, p)$ est soit un nombre, soit un nombre muni d'une constante de répétition (un nombre avec une constante de répétition s'écrit $q * c$, où c est le nombre, q la constante de répétition (entier non signé) qui indique le nombre de répétitions du nombre c). Le nombre d'éléments de la liste doit être égal à celui du tableau. Les nombres contenus dans l'instruction doivent avoir le type défini par le descripteur s_i figurant dans cette instruction. Si un tableau décrit par une instruction de déclaration de type explicite ne contient pas de descripteur de structure, celui-ci doit être contenu soit dans une instruction DIMENSION, soit dans une instruction COMMON (voir plus loin).

On n'attribue pas les valeurs initiales à des variables et des tableaux appartenant à une zone commune non nommée ; on peut attribuer les valeurs initiales à des variables et des tableaux appartenant à une zone commune nommée seulement dans le sous-programme d'initialisation de données (voir p. 8.9.4.4.).

Un programme peut contenir plusieurs instructions de déclaration de type explicites. C'est toujours le type déclaré explicitement qui est retenu. Cela veut dire qu'une instruction explicite annule le type et la longueur établis par une déclaration automatique et une instruction implicite.

Exemple 8.45. La notation || INTEGER * 2 X2/36/, VALUE || est une instruction de déclaration de type explicite. Elle déclare

que les deux variables, X2 et VALUE, sont entières et ont deux octets de longueur chacune. De plus, il attribue la valeur initiale 36 à la variable X2.

EXEMPLE 8.46. La notation || COMPLEX ITEM, D/2.1, 3.08/, E * 16 || est une instruction de déclaration de type explicite qui déclare complexes les variables ITEM, D et E; ITEM et D ont une longueur standard de 8 octets (4 octets pour la partie réelle et autant pour la partie imaginaire). La longueur de la variable complexe E est déclarée de 16 octets (8 octets à chacune des deux parties). La valeur initiale (2.1, 3.08) est attribuée à la variable D.

EXEMPLE 8.47. La notation || REAL * 8I, ALPHA, 12 * 4, ITEM (5,5) || est une instruction de déclaration de type explicite qui déclare réels les variables I, ALPHA, 12 et le tableau ITEM. Les variables I, ALPHA et chaque élément du tableau ITEM ont une longueur de 8 octets. La longueur totale du tableau ITEM est de 200 octets ($5 \times 5 \times 8$). Celle de la variable 12 est de 4 octets.

EXEMPLE 8.48. La notation || REAL A (5,5)/20 * 6.9E2,5 * 1.0/, B (100) /100 * 0.0/ || est une instruction de déclaration de type explicite qui déclare réels les tableaux A et B. Le tableau A a 25 éléments (4 octets par élément); le tableau B a 100 éléments (4 octets par élément). De plus, les 20 premiers éléments du tableau A ont la valeur initiale 6.9E2, les 5 éléments qui restent ont la valeur initiale 1.0. Chacun des 100 éléments du tableau B a la valeur initiale 0.0.

L'*instruction de dimensions de tableaux* a la forme

$$\| \text{DIMENSION } a_1 (k_1),\ a_2 (k_2),\ \ldots,\ a_n (k_n),$$

où a_i ($i = 1, 2, \ldots, n$) est un nom de tableau, k_i ($i = 1, 2, \ldots, n$) un descripteur de structure de tableau ($a_i (k_i)$ est un déclarateur de tableau).

Une instruction DIMENSION ne peut contenir que l'information sur les tableaux dont la structure n'est pas décrite par des instructions de déclaration de type explicites.

L'*instruction de définition de zones communes* a la forme

$$\| \text{COMMON}/r_1/c_1,\ /r_2/\ c_2,\ \ldots,\ /r_n/c_n \|,$$

où r_i ($i = 1, 2, \ldots, n$) est un nom de zone commune; c_i ($i = = 1, 2, \ldots, n$) une liste des grandeurs mises dans la zone commune r_i. *Un nom de zone commune* est soit un identificateur, soit vide. Dans le dernier cas la zone commune est dite *non nommée* ou *blanche*.

Si c_1 se rapporte à une zone commune non nommée, le paramètre r_1 est omis. Si c_i se rapporte à une zone commune non nommée et que $i \neq 1$, alors le paramètre $/r_i/$ doit avoir la forme // (deux barres obliques).

Une liste de grandeurs mises dans une zone commune a la forme $b_1, b_2, \ldots, b_p$, où b_j ($j = 1, 2, \ldots, p$) est soit un identificateur de variable, soit un identificateur de tableau, soit un déclarateur de tableau. Dans le dernier cas b_j a la forme a (k), où a est un identificateur de tableau, (k) son descripteur de structure.

Dans le cas général, un programme FORTRAN se compose d'un programme principal, de plusieurs sous-programmes, ainsi que d'un moniteur et de programmes d'entrée-sortie du système d'exploitation. La mémoire principale s'avère souvent insuffisante pour y mettre tous les sous-programmes à la fois. Il est possible alors de stocker une partie de sous-programmes sur disques ou bandes et de les lire au fur et à mesure de leur appel par le programme principal. Une autre possibilité est d'exécuter les sous-programmes successivement, l'un après l'autre. Dans ce cas, le changement de programmes implique une perte de l'information stockée dans la mémoire. à l'exception des blocs décrits par une instruction COMMON. Un bloc commun est un domaine spécialement réservé aux données communes à deux ou plusieurs programmes, ou bien au programme principal et à ses sous-programmes. Les variables y sont disposées dans l'ordre déterminé par leur succession dans l'instruction de définition de zones communes correspondante.

Les blocs communs de même nom, qui se rapportent à des sous-programmes différents d'un même programme, doivent satisfaire aux conditions suivantes:

— les blocs nommés doivent avoir le même nombre d'éléments (de variables et d'éléments de tableaux);

— les blocs communs blancs peuvent avoir les nombres différents d'éléments (les blocs non nommés sont supposés de même nom);

— le type et la taille des éléments d'un bloc commun doivent être les mêmes que le type et la taille des éléments correspondants des autres blocs communs (correspondance des éléments définis dans les positions identiques du bloc).

Exemple 8.49. Supposons que dans un programme il y ait deux instructions COMMON : || COMMON X, Y, Z, /A1/X1, X2AB,//C, D|| et || COMMON/A1/Z, C2, // X, C1 2B,/B1/X, Y1||.

Conformément à ces instructions, trois blocs communs seront créés: un bloc non nommé où seront placées les grandeurs X, Y, Z, C, D, C12B; un autre, de nom A1, réservé aux grandeurs X1, X2AB, Z, C2; un troisième, de nom B1, qui contiendra les grandeurs X, Y1.

Les blocs communs de même nom, qui se rapportent à des sous-programmes différents, seront superposés au cours d'exécution du programme et une zone commune de la mémoire principale sera réservée aux éléments correspondants de ces blocs.

L'*instruction d'équivalence* permet d'économiser les cases de mémoire principale d'un sous-programme ou d'un programme principal. Il a la forme

$$\| \text{EQUIVALENCE}\ (a_1^{(1)}, a_2^{(1)}, \ldots, a_{n_1}^{(1)}), (a_1^{(2)}, a_2^{(2)}, \ldots, a_{n_2}^{(2)}), \ldots$$
$$\ldots, (a_1^{(k)}, a_2^{(k)}, \ldots, a_{n_k}^{(k)}) \|,$$

où $a_i^{(j)}$ ($j = 1, 2, \ldots, k$; $i = 1, 2, \ldots, n_j$) sont des variables (les indices des variables indicées doivent être des entiers non signés; un élément d'un tableau multidimensionnel est donné comme variable indicée par un nombre qui détermine dans ce cas le numéro d'ordre de l'élément dans le tableau); la notation $(a_1^{(j)}, a_2^{(j)}, \ldots, a_{n_j}^{(j)})$ veut dire qu'un même emplacement de mémoire principale sera affecté à toutes les variables énumérées entre parenthèses.

Il est interdit d'affecter un même emplacement à deux variables appartenant à un bloc commun, tandis que les variables n'appartenant pas à un bloc commun peuvent partager avec une variable qui y appartient son emplacement de mémoire.

8.9.4. Sous-programmes.

8.9.4.1. Instruction-fonction. Une instruction-fonction a la forme

$$f\,(y_1, y_2, \ldots, y_n) = a,$$

où f est un identificateur (un nom de fonction), y_i ($i = 1, 2, \ldots, n$) sont des paramètres formels (variables simples) ; dans la liste $y_1, y_2, \ldots$ $\ldots, y_n$ de paramètres formels on n'admet pas des variables identiques; une même variable peut être utilisée comme paramètre formel de plusieurs instructions-fonctions ; a est une expression (une expression peut contenir les paramètres formels de la fonction f, des variables simples ne figurant pas dans la liste de paramètres formels de la fonction f, des identificateurs de fonction).

La description d'une fonction (type, longueur) ainsi que de ses paramètres formels, se fait de même que pour les variables (voir p. 8.9.3).

Une instruction-fonction ne s'exécute que lorsqu'elle est appelée au moyen de l'identificateur de fonction (voir § 8.4), et cela avec l'estimation des conditions suivantes: a) le nombre des paramètres effectifs de l'identificateur de fonction doit être égal à celui des paramètres formels de l'instruction-fonction; b) la taille et le type des paramètres effectifs doivent être identiques à la taille et au type des paramètres formels correspondants (la correspondance est

établie par ordre de succession des paramètres dans la liste de paramètres).

L'exécution d'une instruction-fonction se réduit au remplacement des paramètres formels figurant dans l'expression a par les paramètres effectifs correspondants de l'identificateur de la fonction ; au calcul de la valeur de l'expression a (on transforme ensuite cette valeur afin qu'elle ait le type et la taille de la fonction f) et à l'attribution de cette valeur à l'identificateur de la fonction.

Les noms des paramètres formels d'une instruction-fonction ne peuvent être inclus ni dans une instruction de définition de zones communes, ni dans une instruction d'équivalence (voir p. 8.9.3).

En tant que paramètre effectif d'une fonction on peut utiliser le nom de n'importe quelle autre fonction ou procédure. Il faut alors inclure dans le programme principal une instruction de sous-programmes extérieurs qui a la forme

$$\| \text{EXTERNAL } a_1, a_2, \ldots, a_n \|,$$

où a_i $(i = 1, 2, \ldots, n)$ sont des noms de fonctions ou procédures externes qu'on doit transmettre, en tant que paramètres effectifs, à certains sous-programmes.

8.9.4.2. Sous-programme fonction. Un *sous-programme fonction* sert à calculer les valeurs d'une fonction. Un sous-programme fonction peut se composer de plusieurs instructions dont la première est une instruction FONCTION, et la dernière une instruction de fin (instruction || END ||). Parmi les autres instructions il ne doit pas y avoir d'instructions FONCTION, SUBROUTINE (voir p. 8.9.4.3), BLOCK DATA (voir p. 8.9.4.4), END ni d'instruction d'arrêt.

Une *instruction-fonction* a la forme

$$\| t \text{ FONCTION } fs\,(y_1, y_2, \ldots, y_n) \|,$$

où t est ou bien vide, ou bien un descripteur du type de fonction (si t est vide, le type de la fonction est déclaré automatiquement), s un descripteur de longueur ; f un identificateur (le nom symbolique de la fonction à définir) ; $y_1, y_2, \ldots, y_n$ une liste de paramètres formels ; y_i $(i = 1, 2, \ldots, n)$ est soit un paramètre formel, soit un paramètre formel mis entre les barres || / || (dans ce cas le paramètre formel est une variable simple). En tant que paramètre formel on peut utiliser une variable simple, un nom de tableau, un nom de fonction ou de procédure externes. Si un paramètre formel est un nom de tableau, le programme doit contenir ou bien une instruction de définition de dimensions de tableaux, ou bien une instruction de type explicite qui contient un descripteur du tableau en question. Les paramètres formels n'ont pas de valeurs initiales et ne peuvent pas faire partie d'instructions de définition de zones

communes, ni d'instructions d'équivalence. Une relation quantitative entre les paramètres formels et effectifs et une correspondance entre les noms de fonctions (de procédures) externes qui représentent les paramètres formels et effectifs sont à satisfaire. La correspondance entre les paramètres formels et effectifs s'établit par ordre de leur succession dans les listes de paramètres. Un paramètre effectif peut être: a) uniquement une variable, si le paramètre formel correspondant est une variable qui prend sa valeur au cours d'exécution du sous-programme, dans le cas contraire on peut utiliser pour paramètre effectif un nombre, une variable et une expression; b) un identificateur de tableau ou une variable indicée (élément de tableau, si le paramètre formel correspondant est un identificateur de tableau. Le nombre d'éléments d'un tableau formel ne doit pas dépasser celui du tableau effectif. Dans les tableaux formels, on admet ce qu'on appelle dimensions ajustables. Le descripteur de structure de tels tableaux contient, en tant que valeurs maximales de certains indices, des variables entières simples; ce sont les dimensions ajustables. Chacune de ces dimensions doit figurer soit dans la liste de paramètres formels du sous-programme fonction, soit dans l'un des blocs communs. Vers le moment d'appel du sous-programme fonction, les variables donnant les dimensions doivent avoir pris certaines valeurs concrètes qui sont dans le premier cas, celles des paramètres effectifs d'identificateurs de fonctions et, dans le deuxième, des variables appartenant aux blocs communs.

Un sous-programme fonction s'exécute après qu'il soit appelé au moyen d'un identificateur de la fonction. L'échange de données (entre le programme appelant et le sous-programme appelé) se réalise de deux façons: en appelant un paramètre effectif par son nom ou par sa valeur. Dans le premier cas, avant l'exécution du sous-programme, le paramètre formel est remplacé partout dans le sous-programme par le paramètre effectif correspondant, et toutes les actions à effectuer sur le paramètre formel seront effectuées en réalité sur le paramètre effectif correspondant. Dans le deuxième cas avant l'exécution du sous-programme, la valeur du paramètre effectif est attribuée au paramètre formel correspondant. Après l'exécution du sous-programme, le paramètre effectif prend la valeur du paramètre formel correspondant. Un paramètre effectif est appelé par son nom, si le paramètre formel correspondant se trouve entre deux symboles || / || ou représente un nom de tableau, de fonction ou de procédure externes; il est appelé par sa valeur, si le paramètre formel représente un nom de variable et n'est pas entre deux symboles || / ||.

Si un paramètre effectif est une expression (i.e. n'est pas une variable), il est remplacé par la variable simple qui est la valeur de cette expression. Le type et la taille de cette variable sont identiques au type et à la taille de l'expression. Dans un sous-programme

la variable — résultat joue le rôle du paramètre effectif correspondant.

L'exécution d'un sous-programme fonction se termine par une instruction de retour ayant la forme || RETURN ||. Un sous-programme peut contenir plusieurs instructions de retour.

Il est inadmissible qu'un sous-programme appelle lui-même d'une façon directe ou indirecte (par d'autres sous-programmes).

Le type et la taille de fonctions externes dans le programme d'appel sont indiqués par les moyens habituels. Les noms de fonctions et de procédures externes qui figurent dans les listes de paramètres effectifs d'identificateurs de fonctions, doivent être cités dans une instruction EXTERNAL du programme d'appel.

8.9.4.3. Sous-programme procédure. Un *sous-programme procédure* est un programme indépendant, avec les variables locales. Il peut se composer de plusieurs instructions FORTRAN dont la première est une instruction de procédure (SUBROUTINE), la dernière étant une instruction de fin (instruction || END ||). Parmi les autres instructions, il ne doit pas y avoir d'instructions FONCTION, SUBROUTINE, BLOCK DATA, END ni d'instructions d'arrêt.

Une instruction de procédure a la forme

$$\| \text{SUBROUTINE } p\ (y_1, y_2, \ldots, y_n) \|,$$

où p est un nom (identificateur) de procédure, $y_1, y_2, \ldots, y_n$ une liste de paramètres formels, y_i ($i = 1, 2, \ldots, n$) est soit un paramètre formel, soit un paramètre formel mis entre deux symboles || / || (alors le paramètre formel est une variable simple). En tant que paramètre formel on peut considérer une variable simple, un nom de tableau, un nom de fonction ou de procédure externes ou le symbole || * ||. Si un paramètre formel est un nom de tableau, alors la procédure doit contenir ou bien une instruction de définition de dimensions de tableaux, ou bien une instruction de déclaration de type explicite avec un descripteur de tableau. Les paramètres formels n'ont pas de valeurs initiales et ne peuvent pas se trouver dans une instruction de définition de zones communes, ni dans une instruction d'équivalence. Une relation quantitative entre les paramètres formels et effectifs, une correspondance entre les noms de fonctions (de procédures) externes qui représentent des paramètres formels et sont à estimer. La correspondance entre les paramètres (formels et effectifs) s'établit par ordre de leur succession dans les listes de paramètres. Un paramètre effectif peut être: a) uniquement une variable, si le paramètre formel correspondant est une variable prenant sa valeur au cours d'exécution du sous-programme procédure, dans le cas contraire le paramètre effectif peut être un nombre, une variable ou une expression; b) un identificateur de tableau

ou une variable indicée (élément de tableau) si le paramètre formel correspondant est un identificateur de tableau et si le nombre d'éléments du tableau formel ne dépasse pas le nombre d'éléments du tableau effectif; c) une expression de la forme ||& *m*||, où *m* est une étiquette d'instruction, si le paramètre formel est le symbole || * ||. Dans les tableaux formels sont admises des dimensions ajustables. Le descripteur d'un tel tableau contient, en tant que valeurs maximales de certains indices, des variables entières simples qui s'appellent dimensions ajustables. Chaque telle dimension doit figurer soit dans la liste de paramètres formels du sous-programme procédure, soit dans l'un des blocs communs. Vers le moment d'appel du sous-programme procédure, les variables fixant les dimensions doivent avoir pris des valeurs concrètes qui sont, dans le premier cas, celles des paramètres effectifs de l'instruction de procédure et, dans le deuxième, celles des variables appartenant aux blocs communs.

Le type et la longueur des paramètres effectifs d'une instruction de procédure doivent être identiques au type et à la longueur des paramètres formels correspondants du sous-programme procédure.

Un sous-programme procédure ne s'exécute que lorsqu'il est appelé au moyen d'une instruction de procédure. Il se produit alors un transfert des données du programme appelant au sous-programme appelé, les entités concernées étant les paramètres formels et effectifs. L'échange de données (entre le programme appelant et le sous-programme appelé) se réalise de deux façons : en appelant les paramètres effectifs par leur nom et par leur valeur. Dans le premier cas, avant l'exécution du sous-programme procédure, un paramètre formel est remplacé partout dans le sous-programme par le paramètre effectif correspondant, et toutes les actions à effectuer sur le paramètre formel seront effectuées, en réalité, sur ce paramètre effectif. Dans le deuxième cas, avant l'exécution du sous-programme, la valeur d'un paramètre effectif est attribuée au paramètre formel correspondant. Après l'exécution du sous-programme, le paramètre effectif prendra la valeur du paramètre formel correspondant. Un paramètre effectif est appelé par son nom si le paramètre formel correspondant se trouve entre deux symboles || / || ou s'il représente un nom de tableau ou un nom de fonction ou de procédure externes; il est appelé par sa valeur, si le paramètre formel correspondant représente un nom de variable et n'est pas entre deux symboles || / ||.

Si un paramètre effectif est une expression (i.e. n'est pas une variable), alors il est remplacé par une variable simple — résultat de calcul de l'expression. Le type et la longueur de cette variable sont identiques au type et à la longueur de l'expression. Dans un sous-programme procédure la variable — résultat joue le rôle de paramètre effectif correspondant.

L'exécution du sous-programme procédure se termine par une instruction de retour ayant la forme || RETURN || ou par une instruction de retour étiquetée qui a la forme || RETURN n ||, où n est une variable simple entière longueur standard ou une constante entière non signée. La première de ces instructions passe le contrôle à l'instruction du programme appelant qui suit l'instruction d'appel du sous-programme procédure, la deuxième n'est utilisée que lorsque la liste des paramètres formels contient un symbole || * ||. L'instruction || RETURN n || passe le contrôle à l'instruction du programme appelant dont l'étiquette est citée dans la liste des paramètres effectifs de l'instruction d'appel du sous-programme procédure et correspond au n-ème astérisque dans la liste des paramètres effectifs de la procédure, à compter de la gauche vers la droite.

EXEMPLE 8.50. Soit un sous-programme procédure

```
||SUBROUTINE ALPHA (X, Y, Z, V)
     Z=(X+Y)**2-X
     V=(X-Y)*X
     RETURN
     END||,
```

où les paramètres formels sont X, Y, Z et V, les deux premiers désignant les données d'entrée, les deux derniers de sortie.

A la suite d'exécution des instructions

```
||X=2.7
CALL ALPHA (6.0, X, Y, Q)||
```

les actions suivantes seront accomplies:

1. Affectation des valeurs 6.0 et 2.7 des premier et deuxième paramètres effectifs aux paramètres formels correspondants X et Y.
2. Calcul des valeurs 69.69 et 19.80 des variables Z et V respectivement.
3. Affectation aux paramètres effectifs Y et Q des valeurs des paramètres formels correspondants. Ainsi, Y aura la valeur 69.69 et Q la valeur 19.80.
4. Sortie du sous-programme procédure selon l'instruction || RETURN || vers l'instruction du programme appelant qui suit immédiatement l'instruction de procédure

|| CALL ALPHA (6.0, X, Y, Q) ||.

EXEMPLE 8.51. Soit un sous-programme procédure:

```
||SUBROUTINE RED (X, Y, *, Q, *)
    IF(X), 1, 2, 2
    1Q=Y
    RETURN 1
    2IF(Y—4)3, 3, 1
    3Q = 6.0
    RETURN 2
    END||.
```

Supposons que l'appel du sous-programme procédure soit réalisé par l'instruction d'appel

|| CALL RED (A, 2.3, & 22, B, & 15||.

Alors, si $A < 0$, le retour aura lieu à l'instruction du programme appelant qui a l'étiquette 22, sinon à l'instruction d'étiquette 15.

Un sous-programme procédure ne doit pas appeler lui-même que ce soit directement ou par d'autres sous-programmes.

Les noms des fonctions ou procédures figurant dans la liste des paramètres effectifs d'une instruction de procédure doivent être mentionnés dans une instruction EXTERNAL du programme appelant.

8.9.4.4. Sous-programme de données. Pour attribuer les valeurs initiales aux grandeurs figurant dans les blocs communs nommés, on utilise un sous-programme de données. Un tel sous-programme peut se composer de plusieurs instructions dont la première une instruction de données ayant la forme || BLOCK DATA||, et la dernière une instruction de fin || END ||. Les autres instructions du sous-programme peuvent être uniquement des instructions de définition de zones communes, de type de dimensions de tableaux et d'équivalence communes. Les instructions de zones sont utilisées pour former des blocs communs nommés. Les valeurs initiales sont attribuées aux grandeurs figurant dans ces blocs par des instructions de déclaration de type explicites.

§ 8.10. Programme FORTRAN

Un algorithme donné dans le FORTRAN (un programme FORTRAN) se présente sous forme d'une suite d'instructions (appelée programme principal) dont la première est une instruction || PROGRAM *a*||, où *a* est un nom (identificateur) de programme,

et la dernière est une instruction de fin (instruction || END ||), et d'un ensemble de sous-programmes (fonctions, procédures, d'initialisation).

Chaque sous-programme, tout comme le programme principal, représente une suite d'instructions dont la première est une instruction « titre » correspondante et la dernière, une instruction de fin.

Les étiquettes d'instructions, ainsi que les noms de variables, de tableaux, et d'autres grandeurs sont locaux au programme principal et à chaque sous-programme ce qui permet d'utiliser les mêmes étiquettes et identificateurs dans différents programmes.

Les branchements entre le programme principal et les sous-programmes ainsi qu'entre les sous-programmes sont réalisés à l'aide d'instructions d'appel correspondantes.

Il est inadmissible que le programme principal appelle lui-même comme sous-programme.

REMARQUE 2. Dans les systèmes de programmation existants qui sont basés sur le FORTRAN, il est recommandé d'écrire les instructions d'un programme dans l'ordre suivant :

1) instruction titre du programme principal (d'un sous-programme) ;

2) instructions de description de fichiers ;

3) instruction de déclaration de type implicite ;

4) instruction de déclaration de type explicite ; instruction de dimensions, instruction de définition de zones communes ;

5) instruction d'équivalence ;

6) instructions de fonction, de procédure ;

7) instruction de format et instructions exécutables (inconditionnelles, conditionnelles, d'entrée-sortie) ;

8) instruction de fin.

CHAPITRE 9

INTRODUCTION AU LANGAGE PL/1

Une version primitive d'un nouveau langage algorithmique élaboré par les chercheurs de la firme IBM et de l'Association des utilisateurs des machines IBM sous le nom abrégé NPL, fut publiée en 1964. Le langage a subi depuis de nombreuses modifications, son nom actuel est PL/1 (Programming Language).

Paru après les langages évolués tels que FORTRAN, ALGOL, le langage PL/1 présente certaines de leurs particularités. Ainsi, il tient de l'ALGOL la notion de bloc de programme, la possibilité d'allocation dynamique de mémoire, les moyens d'appel de procédures (y compris récursives); sa façon de définir les formats de données est analogue à celle du FORTRAN, etc. En même temps, le langage PL/1 a ses propres idées et notions. Il a des possibilités nouvelles qui permettent d'élargir sensiblement son domaine d'application par rapport aux langages algorithmiques cités. On peut signaler, par exemple, l'utilisation des données de types différents (binaires, décimales, complexes, chaînes, etc.), de larges possibilités d'organisation des données (blocs, tableaux, textes, fichiers, etc.), une grande collection de fonctions et de procédures standard, la possibilité d'une exécution asynchrone (parallèle) du programme, etc.

§ 9.1. Alphabets du langage PL/1

Le langage PL/1 possède deux alphabets qui se distinguent par les ensembles de symboles de base. Le premier alphabet contient 60 symboles de base, le deuxième, 48 symboles.

L'alphabet de 60 symboles se compose des chiffres arabes || 0 || 1 || 2 || 3 || 4 || 5 || 6 || 7 || 8 || 9 ||, de 29 lettres (dont 26 lettres latines de A à Z et trois lettres spéciales : signe dollar $, signe prix @ et signe de numéro #) et de 21 symboles spéciaux : || ⊔ || = || + || — || * || / || (||) || , || ; || ' || % || . || : || < || > || | || & || ¬ || — || ? || qui se lisent respectivement comme « blanc », « signe d'égalité », « plus », « moins » « signe de multiplication », « signe de division », « parenthèse ouvrante », « parenthèse fermante », « virgule », « point-virgule »,

« guillemet », « pour cent », « point », « deux points », « signe plus petit que », « signe plus grand que », « symbole ou », « symbole et », « symbole de négation », « symbole de segmentation » (barre au-dessous de la ligne), « point d'interrogation ».

Pour obtenir les symboles de base de l'alphabet de 48 symboles, il faut éliminer de l'alphabet de 60 symboles les symboles de base || ¬ || & || | || > || < || — || # || @ || ? ||. De plus, il faut remplacer les symboles || : || ; || % || par les symboles respectifs || .. ||,. || // ||. On se rappellera en outre que le symbole || .. || remplaçant || : || doit être précédé d'un blanc si le symbole qu'il suit est un point; les deux barres inclinées remplaçant le signe || % || doivent être précédées (suivies) d'un blanc, si le symbole précédent (suivant) est || * ||. Le symbole ||,. || ne peut remplacer || ; || que si le symbole || ; || n'est pas à l'intérieur d'un commentaire ou d'une chaîne de caractères ni immédiatement suivi d'un chiffre.

Certains symboles de base de l'alphabet de 60 symboles sont utilisés comme délimiteurs. On distingue trois types de *délimiteurs*: signes d'opération, parenthèses, séparateurs et autres.

Les *signes d'opérations* sont:

— *les signes d'opérations arithmétiques* || + || — || * || || ** || / || qui se lisent respectivement comme « plus », « moins », « multiplié par », « à la puissance », « divisée par »;

— *les signes de relation* || > || ¬ > || > = || = || ¬ = || < = || < || ¬ < || qui se lisent respectivement comme « est plus grand que », « n'est pas plus grand que », « est plus grand ou égal à », « est égal à », « n'est pas égal à », « est plus petit ou égal à », « est plus petit que », « n'est pas plus petit que » (dans l'alphabet de 48 symboles ces signes de relation ont respectivement la forme || GT || NG || GE || EQ || NE || LE || LT || NL ||);

— *les signes d'opérations logiques* || ¬ || & || | || qui se lisent respectivement comme « non », « et », « ou » (dans l'alphabet de 48 symboles ces signes ont respectivement la forme: || NOT || AND || OR ||);

— *le signe d'opération sur les chaînes* ||$\overset{\times}{\times}$|| (concaténation) *).

Les *parenthèses* sont figurées par les symboles || (||) ||.

On considère comme *séparateurs et autres délimiteurs*:

— les *séparateurs constructifs* ||, || ; || : || = || ⊔ || ' || . || qu'on lit respectivement « virgule », « point-virgule », « deux points », « signe d'affectation », « signe d'espace (ou de blanc) », « guillemet » et « point »;

— le *pour cent* || % ||;

— le *pointeur* || — > ||.

*) Dans les descriptions originales du langage il est désigné par || || ||, i.e. par une double barre verticale.

Comme on a dit plus haut, dans l'alphabet de 48 symboles les « deux points », « point-virgule », « pour cent » et « pointeur » sont représentés autrement.

On utilise dans le PL/1 certains mots de service dont : les descripteurs et les auxiliaires qui précisent les propriétés de procédures et d'autres éléments du langage; les mots séparateurs || THEN || ||ELSE|| BY|| TO|| WHILE||; les noms de fonctions incorporées qui sont les noms des algorithmes prédéfinis dans le langage et accessibles au programmeur; les noms de situations qui déterminent certaines actions spéciales prévues par le programmeur et impliquées par telle ou telle situation ; les noms d'instructions.

§ 9.2. Structures primaires du PL/1

Les structures primaires du PL/1 sont les nombres, les identificateurs et les chaînes.

Nombres. On distingue trois classes de nombres dans le langage PL/1 : réels, complexes et livres.

On considère comme *nombres réels* les nombres entiers (décimaux et binaires), les nombres (décimaux et binaires) en virgule fixe et en virgule flottante.

Une suite finie de chiffres décimaux s'appelle *entier décimal sans signe*. Un entier décimal non signé ou un entier décimal non signé et précédé d'un signe || + || ou || — || s'appelle *entier décimal.*

EXEMPLE 9.1. Les notations || 1001 || 20|| 00|| 0|| 276 || 3|| sont les entiers décimaux non signés; les notations || —0|| +000|| —34|| 25|| 06450|| —1240|| —001768|| sont des entiers décimaux.

Toute suite finie se composant des chiffres || 0|| 1 || s'appelle *entier binaire sans signe.* Pour distinguer un entier binaire d'un entier décimal, le premier est suivi du symbole || B||. Cette remarque concerne tous les types de nombres binaires que nous verrons plus bas.

Un entier binaire non signé, de même qu'un entier binaire non signé précédé d'un signe || + || ou || — || s'appelle *entier binaire.*

Exemple 9.2. Les notations || 0B || 000B|| 01000B|| 010001B|| 111B|| 1B|| sont des entiers binaires non signés; les notations ||—0B||+00B || —0000B|| —100B|| +1010B|| 1000B|| 000B || sont des entiers binaires.

On appelle *fraction régulière décimale* (*binaire*) un entier décimal (binaire) sans signe précédé d'un point. On appelle *nombre décimal* (*binaire*) *non signé à point fixe* une fraction régulière décimale (binaire), ainsi qu'une suite qui comporte un entier décimal (binaire)

suivi d'une fraction régulière décimale (binaire). On appelle *nombre décimal* (binaire) *à point fixe* tout nombre décimal (binaire) non signé ainsi que toute suite se composant d'un signe || + || ou || — || suivi d'un nombre décimal (binaire) non signé à point fixe.

EXEMPLE 9.3. Les notations ||.0860|| .000||.96|| sont des fractions régulières décimales; les notations || .000B|| .010B|| .111B||.110B|| sont des fractions régulières binaires.

Les notations || .34|| 237.26|| 0.9670|| 194.00|| sont des nombres décimaux non signés à point fixe; les notations || .00B|| .10B|| .00111B|| 0.011B|| 1101.01101B|| sont des nombres binaires non signés à point fixe.

Les notations || .670|| —.0347|| +12.0090|| —970.194|| sont des nombres décimaux à point fixe; les notations || .011B|| —.10100B|| 101.11B|| —0101.0110B|| +.1010B|| +111.111B|| sont des nombres binaires à point fixe.

On appelle *exposant* une suite formée par le symbole || E|| suivi d'un entier décimal.

EXEMPLE 9.4. Les notations || E16|| E — 3|| E + 127|| sont des exposants.

On appelle *nombre décimal* (*binaire*) *à point flottant* une ligne formée par un nombre décimal (binaire) à virgule fixe suivi d'un exposant.

EXEMPLE 9.5. Les notations || 137E — 10|| +8.6E4|| —0.0906E — 03|| +8.E12|| sont des nombres décimaux à point flottant; les notations || —101.0010E10B|| 01.0E — 02B|| .10101E + 15B|| +10001E-7B|| sont des nombres binaires à point flottant.

On appelle *nombre imaginaire* une suite formée par un nombre décimal suivi d'un symbole || I ||.

EXEMPLE 9.6. Les notations suivantes sont des nombres imaginaires: || 23I|| —105I|| 0I|| +46I|| .036I|| —976.04I|| +3.27E — — 12I|| —0.896E4I||.

Dans le langage PL/1 on entend par *nombre complexe* une expression de la forme $a + b$I, où a et b sont des nombres réels décimaux et bI, un nombre imaginaire.

On appelle *livre* une suite se composant de trois nombres séparés l'un de l'autre par un point et dont le dernier est suivi du symbole || L||. Le premier de ces nombres est un entier décimal non signé qui indique le nombre de livres; le deuxième est un entier décimal

non signé qui indique le nombre de shillings ; le troisième (le nombre de pence) est un nombre décimal non signé à point fixe dont la partie entière ne dépasse pas 12.

EXEMPLE 9.7. Les notations suivantes sont des livres : || 0.0.3.75L || 3.12.0.L || 876.19.9.L || 0.0.0.L ||.

On appelle *identificateur* toute suite de lettres, chiffres et symboles || $ || @ || # || — || qui commence par une lettre ou par l'un des symboles || $ || @ || # || et qui contient 31 symboles au plus. Un identificateur ne peut pas être terminé par un symbole || — ||.

EXEMPLE 9.8. Les notations suivantes sont des identificateurs : || A || B25-C || $786 || ALPHA || B2-GRAND || # 32-45 || FV # || || @ A1286B # ||.

Chaînes. Une suite quelconque de symboles s'appelle *corps de chaîne* (*valeur de chaîne*). En particulier, une suite vide est un corps de chaîne. On appelle chaîne un corps de chaîne mis entre guillemets. Dans un corps de chaîne un guillemet est représenté par deux guillemets successifs.

Une suite de deux éléments dont le premier est un entier décimal non signé mis entre parenthèses et appelé *constante de répétition* et le deuxième une chaîne, s'appelle *chaîne avec constante de répétition*. La constante de répétition est employée pour ne pas répéter le corps de chaîne qui, sans ce moyen d'abréviation, devrait être écrit un nombre de fois égal à cette constante. Les deux notations sont équivalentes.

On distingue dans le PL/1 deux types de chaînes : chaîne de caractères et chaînes binaires.

On appelle *chaîne de caractères* une chaîne dont le corps se compose de symboles de base ; on appelle *chaîne binaire* une chaîne dont le corps se compose de chiffres || 0 || et || 1 ||. Pour distinguer une chaîne de caractères d'une chaîne binaire, on fait suivre cette dernière du symbole || B ||.

Exemple 9.9. Les notations suivantes sont des corps de chaînes : || 03AB || 0 || $24ABLC || +/@, + —24.3,A || 011211 || 011 || ; les notations || '2— # BA' || 'AL"27OR' || '011101' || '$ 24FS' || sont des chaînes ; les notations || (4) '2ARF' || (3)'ALPHA' || (10)'A' || sont des chaînes avec constantes de répétitions, elles sont équivalentes respectivement à || '2ARF2ARF2ARF2ARF' || 'ALPHAALPHAALPHA' || ||'AAAAAAAAAA' ||.

Toutes les chaînes citées sont des chaînes de caractères.

Les notations || '011011011'B || '0'B || '1'B || '0000'B || '11'B || sont des chaînes binaires ; les notations || (2)'01110'B || (7)'11'B || (6)'000'B ||

sont des chaînes binaires avec constantes de répétition, elles sont équivalentes respectivement à || '0111001110'B || '11111111111111'B || '000000000000000000'B ||.

§ 9.3. Variables. Tableaux. Structures

On distingue deux types de variables : variable simple et variable indicée. Une *variable simple* représente une grandeur qui prend des valeurs numériques, logiques ou peut être une chaîne et qui est désignée par un identificateur. Une variable est dite indicée si son nom est suivi d'un ou plusieurs indices placés entre parenthèses.

On appelle *tableau* l'ensemble des variables indicées de même nom. Ces variables sont définies simultanément grâce à des déclarations qui fixent le type de variable, le nombre de dimensions et l'intervalle de variation de chaque indice. Elles sont appelées *éléments de tableau.*

Tous les éléments d'un tableau sont de même type et s'écrivent avec le même nombre d'indices, c'est-à-dire s'ils sont des nombres, il faut qu'ils soient en même base, aient la même forme de représentation, le même nombre de chiffres, la même construction (voir le p. 9.6.1). Dans le cas où les éléments du tableau sont des chaînes, ils doivent être tous de même longueur (si les longueurs sont fixes) ou de même longueur maximale (si les longueurs sont ajustables).

L'identificateur commun à tous les éléments d'un tableau s'appelle *identificateur de tableau.* Un tableau est désigné par son identificateur suivi d'une liste d'indices entre parenthèses (qui se présentent sous la forme d'une expression, voir le § 9.5). Grâce à l'utilisation de variables indicées on peut appeler n'importe quel élément de tableau. Avant d'appeler un élément de tableau, les valeurs de chaque expression en indice sont calculées et transformées en entiers. Le nombre d'indices dans la description d'un tableau s'appelle *dimension du tableau.*

EXEMPLE 9.10. Considérons le cas le plus simple où chaque expression en indice est un entier non signé. Alors un tableau de nom AL (3, 4) peut être considéré comme l'ensemble des éléments AL (1,1), AL (1, 2), AL (1,3), AL (1,4), AL (2, 1), AL (2, 2), AL (2, 3), AL (2, 4), AL (3, 1), AL (3,2), AL (3, 3), AL (3, 4).

Ce tableau a deux dimensions, la première ayant pour bornes 1 et 3, la deuxième, 1 et 4.

Etant donné un tableau $x\,(y_1, y_2, \ldots, y_n)$, où x est un nom de tableau, y_i $(i = 1, 2, \ldots, n)$ des expressions en indice, son sous-tableau de nom $x\,(z_1, z_2, \ldots, z_n)$, où z_i $(i = 1, 2, \ldots, n)$ est soit un symbole || * ||, soit coïncide avec y_i, s'appelle *section du tableau* (cross-section). Lorsque les symboles || * || dans la liste

d'indices donnée sont au nombre k ($k \leqslant n$), et que z_j soit le symbole || * ||, alors la section du tableau comprend tous les éléments x (y_1, . . ., y_n) du tableau, obtenus en faisant varier le j-ième indice entre ses bornes. La dimension de la section est égale au nombre de symboles || * || dans la liste d'indices, i.e. au nombre k.

Exemple 9.11. Dans le tableau AL (3,4) de l'exemple 9.10, section AL (*, 4) représente le vecteur (sous-tableau): AL (1,4), AL (2,4), AL (3,4).

Une construction de la forme n_1x_1, $n_2x_2y_2$, . . ., $n_kx_ky_k$ s'appelle structure. Ici n_i (i = 1, . . ., k) sont des nombres décimaux non signés qu'on appelle *niveaux d'hiérarchie* (n_1 est inférieur à tous les autres niveaux); x_i (i = 1, 2, . . ., k) sont des identificateurs (x_1 est un nom de structure); chaque y_i (i = 2, 3, . . ., k) est vide ou représente une liste d'indices (l'identificateur correspondant x_i s'il n'est pas un nom de structure détermine dans le premier cas le nom d'une variable, dans le deuxième cas le nom d'un tableau dont le nombre de dimension vaut le nombre d'indices dans la liste d'indices). Si n_1 est vide ou égal à 1, il s'agit d'une *structure majeure*, sinon la structure est *mineure*.

Exemple 9.12. La notation || 1RER, 2X, 2Y, 3A, 3B, 4X1, 4Y1, 2Z, 3L1 || définit une structure de nom RER. Les éléments de la structure sont les variables || X || A || X1 || Y1 || L1 || et les structures || 2Y, 3A, 3B, 4X1, 4Y1 ||, ||3B, 4X, 4Y1 ||, || 2Z, 3L1 || qui sont intérieures par rapport à la structure donnée.

La structure donnée peut être mise sous la forme

```
||RER,                                  || 1RER,
     X,                                      2X,
     Y,                                      2Y,
       A, ou, ce qui est le même,              3A,
       B,                                      3B,
         X1,                                     4X1,
         Y1,                                     4Y1,
     Z,                                      2Z,
         L1 ||                                 3L1||.
```

Une telle représentation d'une structure s'appelle parfois forme de la structure.

La notation || RER, 4X, 4Y, 7A, 7B, 8X1, 8Y1, 4Z, SL1 || définit une structure de même forme et avec le même ensemble d'éléments que la structure précédente. La différence est dans la numérotation de niveaux d'hiérarchie.

La notation || 1F, 3ALPHA, 2BETA|| définit une structure de la forme

|| F,
  ALPHA,
  BETA ||.

Le nom de la structure est l'identificateur F, les éléments sont les variables ALPHA, BETA.

La notation || 1X, 2PEN, 3BOOK (2), 2PENCIL|| est une structure de nom X. Ses éléments sont une variable PENCIL, un tableau unidimensionnel BOOK (2) et une structure || 2PEN, 3BOOK (2)||. La structure considérée a la forme

|| X,
  PEN,
    BOOK (1),
    BOOK (2),
  PENCIL ||.

On appelle un élément d'une structure ne contenant pas de tableaux au moyen d'un *nom qualifié* qui représente une suite de noms séparés par des points, de variables de la structure choisies dans des niveaux d'hiérarchie différents de façon à définir un « avancement » hiérarchique univoque à partir du nom de la structure jusqu'à l'élément désigné.

Exemple 9.13. On peut appeler l'élément L1 de la première structure de l'éxemple 9.12 au moyen du nom qualifié RER.Z.L1; l'élément B sera appelé au moyen du nom qualifié RER.Y.B.

Une construction de la forme

$$n_1x_1y_1, \; n_2x_2y_2, \; \ldots, \; n_kx_ky_k$$

qui diffère de la précédente par le fait que y_1 peut être non vide, donc représenter une liste d'indices, définit un *tableau de structures*. Ici x_1 est un nom du tableau de structures, y_1 détermine la dimension du tableau de structures.

Exemple 9.14. Considérons un tableau de structures défini par la notation || 1LIT (3), 2F, 2Z, 3H, 3S||. L'identificateur LIT est le nom du tableau. Les éléments de structure sont les tableaux de noms F, H, S et la structure || 2Z, 3H, 3S||.

Le tableau considéré a la forme

```
|| LIT F(1),
        Z(1) H(1),
             S(1),
   LIT F(2),
        Z(2) H(2),
             S(2),
   LIT F(3),
        Z(3) H(3),
             S(3) ||.
```

La construction de la forme || 1X, 2Y, 2Z (2), 3P (2,2), 3Q (2), 2R || et une structure de nom X ayant pour éléments : les variables Y, R ; les tableaux P (2,2), Q (2) ; le tableau de structures || 2Z (2), 3P (2,2), 3Q (2). ||La structure considérée a la forme

```
|| X,
     Y,
       Z(1),
            P(1, 1, 1),
            P(1, 1, 2),
            P(1, 2, 1),
            P(1, 2, 2),
            Q(1),
       Z(2),
            P(2, 1, 1),
            P(2, 1, 2),
            P(2, 2, 1),
            P(2, 2, 2),
            Q(2),
     R                 ||.
```

L'appel d'un élément d'une structure contenant des tableaux se fait ou bien à l'aide d'un nom qualifié, si le chemin hiérarchique, menant du nom de la structure jusqu'à l'élément donné, répondant à ce nom qualifié, ne passe pas par les tableaux de la structure, ou bien à l'aide de nom qualifié indicé. On appelle *nom qualifié indicé* une suite de noms de variables et de noms de variables indicées séparés par des points. L'ordre de succession des noms est le même que pour un nom qualifié. La liste d'indices suivant un nom indicé se rapporte aux dimensions liées à ce nom, si le dernier nom dans la

structure est celui d'un tableau. Sans modifier l'ordre d'indices, on peut les déplacer vers la gauche ou vers la droite et les associer aux noms de niveau inférieur ou supérieur respectivement. En appelant des éléments d'une structure on peut omettre les noms de structures sans indices, à condition que les noms qui restent permettent d'atteindre l'élément voulu.

EXEMPLE 9.15. Soit A un tableau de structures ayant la forme ||1A (10, 12), 2B (5), 3C (7), 3D||. Supposons qu'il faut appeler le septième élément du tableau C contenu dans le cinquème élément du tableau B qui, à son tour, se trouve à l'intersection de la dixième ligne et de la douzième colonne du tableau A. On peut le faire à l'aide de l'un des noms qualifiés suivants: || A (10, 12). B (5). C (7)|| A (10), B (2,15).C(7)|| A(10).B (12).C (5,7)|| A.B (10,12,5).C (7)|| A.B (10, 12).C (5, 7)|| A.B (10).C (12, 5, 7)|| A.B.C (10, 12, 5, 7)|| A (10. 12), B.C (5, 7)|| A (10).B.C (12, 5, 7)|| A (10, 12, 5, 7).B.C||.

L'appel univoque de cet élément peut être réalisé à l'aide de l'un des noms qualifiés || B (10, 12, 5).C (7)|| B (10, 12).C (5, 7)|| B (10).C (12, 5, 7)|| B.C (10, 12, 5, 7)|| qui ne contiennent pas le nom A, ainsi qu'à l'aide des noms || A.C (10, 12, 5, 7)|| A (10, 12).C (5, 7)|| A (10).C (12, 5, 7)|| A (10, 12, 5, 7).C|| qui ne contiennent pas le nom B, etc.

Par leur destination, les variables sont classées en *variables arithmétiques* et *variables de contrôle.*

Il existe trois modes de description d'une variable, d'un tableau ou d'une structure; une description explicite, une description contextuelle, une description implicite.

Pour une *description explicite* on se sert de l'instruction ||DECLARE|| (voir le p. 9.7.17). La description explicite des variables groupées en fichiers se fait au moyen de l'instruction || OPEN|| (voir le p. 9.7.18.1).

Il s'agit d'une *description contextuelle* lorsque, sans décrire complètement une variable, on est en mesure de définir certaines de ses caractéristiques (attributs) en partant ou bien d'autres caractéristiques de cette variable, ou bien de la manière dont on l'utilise. Par exemple, sans mentionner explicitement qu'une variable donnée est un nom d'entrée, on peut reconnaître qu'elle est un nom d'entrée dans une procédure d'après l'instruction d'appel de la procédure ou, puisque cette variable est un nom de fonction, d'après l'instruction d'appel d'une fonction avec une liste non vide d'arguments. Un autre exemple : sans mention explicite du fait que X est une variable pointeur, on peut le découvrir si une description explicite d'une variable Y utilise le descripteur || CONTROLLED (X)|| (voir le p. 9.6.3).

Une *description implicite* apparaît, par exemple, dans le cas où une variable, non déclarée de façon explicite ou contextuelle, représente une expression contenant des variables déclarées.

Les variables arithmétiques sont classées en quatres types: les variables *entières*, *réelles*, *complexes* et *chaînes*. Chaque variable ne peut prendre que des valeurs correspondant à son type. En outre, la valeur d'une variable peut être représentée dans l'un des systèmes de numération : binaire ou décimal, et pour les trois premiers types, en forme point fixe ou point flottant.

Pour décrire le type, la base de numération et la forme de représentation des variables, on se sert des descripteurs respectifs

|| INTEGER || REAL || COMPLEX || CHARACTER ||;

|| BINARY || DECIMAL ||; ||FLOATE || FIXED || (voir le p. 9.6.1).

Les variables de contrôle sont de *type étiquette*, *branche*, *événement*, *pointeur*, *domaine*, *cellule*.

On peut étiqueter les instructions (voir le § 9.7) d'un programme, i.e. leur donner des noms individuels, appelés *étiquettes*. Chaque étiquette est ou bien un identificateur, ou bien une variable de type étiquette dont toutes les valeurs sont des étiquettes (identificateurs). Les variables de type étiquettes peuvent être réunies en des tableaux ou des structures.

Une étiquette et une variable de type étiquette sont appelées *éléments de type étiquette*. Pour les décrire on se sert du descripteur ||LABEL|| (voir le p. 9.6.1).

Une étiquette d'une instruction || PROCEDURE || ou || ENTRY || se déclare explicitement en tant que nom d'entrée.

Le langage PL/1 prévoit deux façons d'exécution d'un programme : synchrone et asynchrone.

L'*exécution synchrone* est une exécution successive des instructions dans l'ordre de leur notation. L'ordre successif est modifié par les instructions suivantes: || RETURN || CALL || END || SIGNAL || GO TO || PROCEDURE || STOP || IF ||.

L'*exécution asynchrone* est une exécution simultanée de différents groupes d'instructions. Ce mode d'exécution s'applique à un programme *branchu*. Une *branche* est un ensemble d'instructions de l'algorithme exécutées successivement l'une après l'autre. Une instruction peut appartenir à plusieurs branches. Pour appliquer ce mode il faut que le programme possède au moins une branche. Parmi toutes les branches on choisit une *branche principale*. Si le programme s'exécute de manière synchrone, la branche principale coïncide avec le programme même. En mode asynchrone, la branche principale est la première procédure extérieure (voir le p. 9.7.9). A toute branche, à l'exception de la principale, ou peut attribuer un nom qui peut être utilisé pour définir l'hiérarchie des branches.

Un *nom de branche* est un identificateur. Une branche peut également être définie par une variable qu'on appelle *variable de type branche*. Cette variable peut être élément d'un tableau ou d'une structure. La description d'une variable du type branche se fait au moyen d'un descripteur du type || TASK || (voir le p. 9.6.1). On forme une branche en complétant l'instruction || CALL || (voir le p. 9.7.11) par l'une au moins des clauses || TASK || EVENT || PRIORITY ||.

La procédure appelée sera alors exécutée parallèlement à la procédure appelante.

La clause || TASK || dans l'instruction || CALL || est spécifiée dans le cas où il faut attribuer un nom à une branche. On a besoin d'un nom de branche pour définir ou modifier la priorité de la procédure (branche) appelée, puisque la fonction || PRIORITY || (voir le § 9.9.) a un nom de branche pour argument.

La clause || EVENT || est spécifiée dans le cas où il faut synchroniser deux exécutions parallèles, par exemple, lorsque les résultats de calcul d'une branche doivent être utilisés dans une autre. On fait ceci au moyen de l'instruction || WAIT || (voir le p. 9.7.16) qui arrête temporairement l'exécution d'une branche jusqu'à ce que certains événements ne se produisent, par exemple une division par le zéro, un dépassement de capacité, etc. Un événement peut être donné par son nom qui est un identificateur ou une variable du type événement. Une variable du type événement peut être élément d'un tableau ou d'une structure.

A une variable de type événement est lié le statut d'achèvement (|| '0'B || signifie « non achèvement », || '1'B || « achèvement »). Si une variable du type événement est liée à une branche, le statut d'achèvement de cette variable correspondra à celui de la branche. On établit le statut d'achèvement d'une variable du type événement en introduisant la pseudo-variable || EVENT || (voir le p. 9.7.2, le § 9.9). La description d'une variable du type événement se fait à l'aide du descripteur || EVENT ||.

La priorité d'une branche est définie par l'une des méthodes suivantes: a) en spécifiant la clause || PRIORITY || dans l'instruction || CALL ||; b) en affectant à la branche la pseudo-variable || PRIORITY || avant l'exécution de l'instruction || CALL || qui crée la branche.

Une branche peut être achevée dans l'un des cas suivants: a) le contrôle dans la branche donnée est passé à une instruction || STOP || (voir le p. 9.7.7); b) le contrôle dans la branche donnée est échu à une instruction || RETURN || (voir le p. 9.7.12); c) le contrôle dans la branche est passé à une instruction || END || (voir le p. 9.7.13).

Les notions de pointeur, de domaine et de cellule sont liées au problème de gestion de la mémoire.

Dans le cas général, la place d'un élément de données dans la mémoire est déterminée au moment d'allocation de la mémoire.

Pourtant, le langage PL/1 dispose d'un moyen, appelé *pointeur*, d'identifier la place d'une variable dans la mémoire. Une variable placée dans la mémoire au moyen d'un pointeur et déclarée avec || CONTROLLED || est appelée *variable basée* (voir le p. 9.6.3). De telles variables ne s'utilisent que pour le traitement des listes ou avec || RECORD || (voir le p. 9.8.1)

Un nom de pointeur est un identificateur ou une variable de type pointeur. Cette dernière peut être élément d'un tableau ou d'une structure. On peut la décrire par un descripteur || POINTER || (voir le p. 9.6.3).

Le langage PL/1 prévoit une possibilité de stocker les données dans les parties de la mémoire identifiées par un identificateur ou une variable. On distingue deux types de telles parties. Dans une partie du premier type on peut placer les valeurs de plusieurs variables à la fois, dans une partie du deuxième type, les valeurs d'une seule variable. Donc une partie du premier type (resp. un identificateur ou une variable) est appelée *domaine* (resp. *variable du type domaine*), une partie du deuxième type est appelée *cellule* (resp. *variable du type cellule*).

Une variable du type domaine est décrite par le descripteur || AREA ||, une variable du type cellule, par un descripteur || CELL || (voir le p. 9.6.3). Une variable du type domaine peut être élément d'un tableau ou d'une structure.

§ 9.4. Indicateurs de fonctions

En plus de signes d'opérations qui sont des symboles de base, on utilise dans le PL/1 quelques signes d'opérations spéciaux appelés *indicateurs de fonctions*. Les opérations définies par un indicateur de fonctions s'exécutent en exécutant l'algorithme correspondant. Les notations désignant les valeurs des arguments d'une fonction s'appellent *paramètres effectifs*. Ils peuvent être nombres, chaînes variables, tableaux, structures, expressions, noms d'entrées en des procédures.

Un indicateur de fonctions s'écrit comme un identificateur suivi d'une liste d'arguments entre parenthèses (de paramètres effectifs) séparés par des virgules. L'identificateur est un nom d'entrée dans la procédure (voir le p. 9.7.9).

Le langage PL/1 possède une grande bibliothèque de fonctions incorporées (standard) parmi lesquelles il y a des fonctions arithmétiques, des fonctions traitant les chaînes, les tableaux ou les structures, des fonctions de contrôle d'interruptions, des fonctions de traitement des listes (voir le § 9.9).

§ 9.5. Expressions

9.5.1. Expression arithmétique. On appelle *expression arithmétique*: a) un nombre ; b) une variable; c) un indicateur de fonction; d) une expression arithmétique entre parenthèses; e) une expression arithmétique précédée d'un signe || — || ou || + ||; deux expressions arithmétiques liées par l'un des signes || + || — || * ||/|| || * * ||.

Les opérandes d'une expression peuvent différer par le type, la forme de représentation et la base de numération. Alors les règles suivantes de transformation des opérandes sont à estimer: IO

1) Lorsque les opérandes sont de types différents (l'un est réel et l'autre complexe), l'opérande réel est mis sous une forme complexe (en y associant une partie imaginaire nulle dont la forme de représentation, la base du système de numération et le nombre de chiffres sont les mêmes que pour la partie réelle). Pourtant, lorsqu'il s'agit d'une élévation à une puissance, où le deuxième opérande est à point fixe, celui-ci reste inchangé.

2) Lorsque la forme de représentation des opérandes est différente, l'opérande fixe est transformé en flottant, à l'exception de l'élévation à une puissance, où le premier opérande est flottant et le deuxième fixe. Dans ce cas le deuxième opérande ne se transforme pas.

3) Lorsque les bases de numération sont différentes, l'opérande décimal est converti en binaire.

L'opération arithmétique s'exécute dès que les transformations décrites sont effectuées. Le type, la forme de représentation, la base de numération et le nombre de chiffres du résultat sont ceux des opérandes. Dans le cas d'une opération à deux places, la longueur du résultat est égale à celle du plus long des opérandes, si ces derniers sont à point flottant. Lorsqu'ils sont à point fixe, le nombre de chiffres du résultat est défini selon les règles suivantes.

3. En addition et soustraction:

$$m = \min (N, \max (p - q, r - s) + \max (q, s) + 1);$$
$$n = \max (q, s) \ ^{*}).$$

2) En multiplication:

$$m = \min (N, p + r + 1); \ n = q + s,$$

3) En division:

$$m = N; \ n = N - p + q - s.$$

---

*) Ici et plus loin m est le nombre de chiffres du résultat, n le facteur d'échelle du résultat, p le nombre de chiffres du premier opérande, q le facteur d'échelle du premier opérande, r le nombre de chiffres du deuxième opérande, s le facteur d'échelle du deuxième opérande, N la longueur maximale du nombre dans la version adoptée du langage.

4) En exponentiation:

$$m = E\,[(p + 1) * y - 1], \quad n = E\,[p * y],$$

où $E(z)$ est la partie entière du nombre z, y est le deuxième opérande ($y \neq 0$) représenté par un réel non signé à point fixe. Si y n'est pas un réel à point fixe, alors le premier opérande est converti en flottant, et l'opération s'effectue d'après les règles qui interviennent dans le cas des nombres réels à point flottant.

Les règles 1 à 4 sont valables pour les nombres réels et complexes.

Les signes d'opérations arithmétiques dans le PL/1 sont compris au sens usuel, à l'exception des cas suivants:

1. L'opération $x_1 * * x_2$ d'élévation à une puissance à opérandes réels x_1 et x_2, n'est pas définie si $x_1 \leqslant 0$ et $x_2 \leqslant 0$.

2. L'opération d'élévation à une puissance à opérandes complexes z_1 et z_2: a) a le résultat nul lorsque $z_1 = 0$ et le nombre z_2 a une partie réelle positive et une partie imaginaire nulle; b) est inadmissible, lorsque $z_1 = 0$ et que la partie réelle de z_2 est au plus nulle ou la partie imaginaire de z_2 n'est pas nulle.

La valeur d'une expression arithmétique est calculée compte tenu des règles usuelles de l'hiérarchie des opérations et des parenthèses. On adopte l'ordre suivant des opérations (décroissant): 1) calcul de la valeur d'un' indicateur de fonction; 2) || * *||, opérations à une place || + || et || — ||; 3) || * |||| / ||; 4) opérations à deux places || + || — ||.

Si deux opérations de même niveau hiérarchique (à l'exception de l'opération || * * ||, suivent l'une l'autre dans une expression, elles sont exécutées de la gauche vers la droite. Les opérations d'exponentiation qui suivent l'une l'autre dans une expression sont effectuées de la droite vers la gauche.

9.5.2. **Expressions chaînes.** On appelle *expression chaîne*: a) une chaîne de caractères; b) deux expressions chaînes liées par le symbole $\left\| \begin{smallmatrix}\times\\\times\end{smallmatrix} \right\|$.

Le signe $\left\| \begin{smallmatrix}\times\\\times\end{smallmatrix} \right\|$ désigne l'opération de concaténation de chaînes: on peut la représenter comme $x \begin{smallmatrix}\times\\\times\end{smallmatrix} y$, où x et y sont des chaînes de caractères ou des chaînes binaires. Cette opération engendre une nouvelle chaîne dont le corps s'obtient en complétant à droite le corps de la chaîne x par celui de la chaîne y. La longueur du résultat vaut la somme des longueurs des chaînes x et y.

EXEMPLE 9.16. Si A est une chaîne de caractères || 'X +/, YZN23' ||, et B une chaîne de caractères ||'SR — K'||, le résultat de l'opération de concaténation A $\begin{smallmatrix}\times\\\times\end{smallmatrix}$ B sera || 'X +/, YZN23SR — K' ||; de même

le résultat de l'opération $A \overset{\times}{\times} B \overset{\times}{\times} C$ sera || 'X +/, YZN23SR — — KPQT'||, si C est une chaîne de caractères || 'PQT'||.

Si A est une chaîne binaire ||'0111011101'B||, et B une autre chaîne binaire || '1001'B||, alors $A \overset{\times}{\times} B$ sera une chaîne binaire ||'01110111011001'B||.

Si les opérandes (chaînes) d'une concaténation sont de types différents (chaîne de caractères et chaîne binaire) alors, avant d'exécuter l'opération, on transforme les chaînes binaires en chaînes de caractères selon les règles données dans 9.5.3.

9.5.3. Expression logique. On appelle *expression logique*: a) une expression logique primaire (chaîne binaire, variable logique, i.e. une variable dont les valeurs possibles sont || '0'B|| et || '1'B||, indicateur d'une fonction logique, i.e. d'une fonction dont les valeurs possibles sont || '0'B|| et || '1'B||, relation, expression logique entre parenthèses); b) une expression logique précédée du signe || ¬ ||; c) deux expressions logiques liées par l'un des signes || & || ou || | ||.

Les opérations « non » (signe d'opération || ¬ ||), « ou » (signe d'opération || | ||), « et » (signe d'opération || & ||) s'exécutent digit à digit. Si les opérandes d'une opération à deux places « et » (« ou ») sont de tailles différentes, le plus court d'entre eux est complété à droite par zéros jusqu'à avoir la taille de l'autre. A la suite de l'opération chaque digit prend une valeur selon la table 9.1.

Table 9.1

Résultats des opérations logiques

A	B	¬A	A & B	A | B
1	1	0	1	1
1	0	0	0	1
0	1	1	0	1
0	0	1	0	0

EXEMPLE 9.17. Soient A une chaîne binaire || '011101'B|| et B une chaîne binaire || '1110'B||; alors ¬ A est une chaîne binaire || '100010'B||, || A & B|| une chaîne binaire || '011000'B||, A | B une chaîne binaire || '111101'B||.

On appelle *relation* un couple d'expressions (arithmétiques, chaînes et binaires) liées par un signe d'opération de relation: || <|| ¬<|| <=|| = || ¬ =|| > =|| >|| ¬ >||.

On distingue trois types de relations: algébriques, chaînes et binaires.

Une relation est dite *algébrique* lorsque ses opérandes sont des expressions arithmétiques. Seules les relations || = || ¬ = || peuvent avoir des opérandes complexes.

Une relations est dite *chaîne* si ses opérandes sont des chaînes de caractères. Une relation chaîne est toujours une comparaison, menée de la gauche vers la droite, des caractères des chaînes des opérandes dans les positions respectives selon leur hiérarchie alphabétique.

Exemple 9.18. Les relations suivantes sont des relations chaînes:

|| 'AB, C' = 'XY, C' || '+CD/I' ¬ = 'A * ZBCI' || '011' > '02760' ||.

Si les opérandes comparés sont de longueurs différentes, le plus court d'entre eux est complété à droite par des blancs.

Une relation est dite *binaire* si les opérandes sont des suites binaires. Une relation binaire signifie une comparaison de la gauche vers la droite, des chiffres binaires dans les positions respectives. Si les opérandes sont de longueurs différentes, le plus court est complété à droite par des zéros.

Exemple 9.19. Les relations suivantes sont binaires:

|| '011101' B < '111011 'B || '110'B ¬ < '000'B ||.

Le résultat d'une comparaison (algébrique, chaîne ou binaire) est une suite binaire de longueur unité, ayant pour valeur || '1'B || si la relation est vraie, et || '0'B || si elle est fausse.

Lorsqu'on compare les opérandes de nature différente, les suites binaires se transforment en chaînes de caractères, celles-ci sont converties en nombres, les nombres à point fixe en nombres à point flottant, les nombres décimaux en nombres binaires, les nombres réels en nombres complexes.

Les conversions de types se font d'après les règles suivantes:

— une suite binaire se transforme en une chaîne de caractères. Le bit 1 se transforme en le caractère 1 et le bit 0 en le caractère 0. La longueur ne change pas. Une suite vide de bits devient une chaîne de caractères vide;

— une chaîne de caractères se transforme en une suite binaire. Les caractères 1 et 0 deviennent les bits 1 et 0 respectivement. La conversion est impossible si la chaîne de caractères contient des caractères autres que 0 et 1. Une chaîne de caractères vide devient une suite binaire vide;

— une chaîne de caractères se transforme en un nombre. Le corps de la chaîne à transformer doit représenter ou bien un nombre

(entier, réel ou complexe), ou bien un nombre précédé et suivi de blancs || ⊔ ||.

9.5.4. Expression scalaire. Les expressions arithmétiques, chaînes et logiques forment la classe des expressions scalaires, et le type particulier de l'expression est déterminé par les opérations qui en font parties: arithmétiques, logiques ou des concaténations. Les variables de types étiquette, domaine, branche, événement ne peuvent se rencontrer dans une expression scalaire qu'en tant qu'arguments de fonctions. Les seules opérations possibles sur ces données du type pointeur sont celles de comparaison ||=|| ¬ =||.

9.5.5. Expressions de type tableaux. Une expression est considérée comme expression de type tableau, lorsque parmi ses opérandes il y a des tableaux ou des structures. Une expression de ce type a pour valeurs des tableaux. Toutes les opérations sur les tableaux s'exécutent sur les éléments respectifs. Pour cette raison, tous les tableaux figurant dans une expression doivent avoir les mêmes bornes.

Exemple 9.20. Si A est le tableau des éléments 3, 4, 5, —3, 2, —4 et B le tableau des éléments 7, 4, —3, 2, 7, 5 alors:

a) || —A|| est le tableau des éléments —3, —4, —5, 3, —2, 4;

b) || 4 * A|| est le tableau des éléments 12, 16, 20, —12, 8, —16;

c) ||A + B|| est le tableau des éléments 10, 8, 2, —1, 9, 1;

d) || A * B|| est le tableau 21, 16, —15, —6, 14, —20;

e) || MAX (A + B, A * B)|| est le tableau 21, 16, 2, —1, 14, 1.

Si une expression de type tableau contient comme opérandes des structures, alors tous les opérandes tableaux de cette expression doivent être tableaux de structures, et toutes les structures figurant dans l'expression doivent avoir la même construction.

Exemple 9.21. Soit X le tableau de structures || 1X (5), 2Y, 2Z||, et A la structure || 1A, 2B, 2C||. Alors l'expression || X + A|| a un sens, elle a pour résultat l'addition de la structure A à chaque structure du tableau X. Cette expression est donc équivalente à la suivante:

|| X (1).Y + A.B
X (1).Z + A.C
X (2).Y + A.B
X (2).Z + A.C
— — — — — — —
X (5).Y + A.B
X (5).Z + A.C ||.

9.5.6. Expressions de type structures. Ce sont des expressions qui ont pour opérandes des structures d'une même construction. Cela signifie que les structures doivent contenir le même nombre de variables et de tableaux en des positions respectives, et les tableaux homologues doivent avoir la même dimension et les mêmes bornes. L'uniformité du type des données n'est pas exigée.

Une expression de type structure engendre une structure. Toutes les opérations effectuées sur les structures sont effectuées sur des éléments respectifs.

EXEMPLE 9.22. Considérons deux structures:

‖ 1 A,	‖ 1 B,
2BETA1,	2BETA1,
3SOS1,	1SOS1,
3SOS2,	1ALPHA,
3SOS3,	1SOS2,
2BETA2,	2BETA2,
3SOS4,	3ALPHA,
3ROT,	3SOS4,
3SOS5 (3) ‖,	3SOS5 (3) ‖.

L'expression ‖ A + 3 * B ‖ représente une expression de type structure et est une forme condensée de notation des expressions suivantes:

‖ A.SOS1 + 3 * B.SOS1,
A.SOS2 + 3 * B.BETA1.ALPHA,
A.SOS3 + 3 * B.SOS2,
A.SOS4 + 3 * B.BETA2.ALPHA,
A.SOS5 + 3 * B.SOS5,
A.ROT + 3 * B.SOS4 ‖,

chacune d'elles étant une expression de type tableau.

§ 9.6. Description des données

On décrit les données au moyen des descripteurs qui sont classés comme suit: descripteurs de données, descripteurs de nom d'entrée, descripteurs d'allocation de mémoire, descripteurs de domaine d'action, descripteurs des fichiers.

9.6.1. Descripteurs de données. On considère comme descripteurs de données les descripteurs de base de numération, de forme de représentation des valeurs, de types de grandeurs, de précision, le descrip-

teur de format, le descripteur de lignes, le descripteur d'étiquette, le descripteur de branche, le descripteur d'événement, le descripteur de dimension, les descripteurs || SECONDARY || ABNORMAL || || REDUCIBLE ||IRREDUCIBLE || USES || SETS ||.

Le *descripteur de base de numération* indique si la valeur d'une grandeur est binaire ou décimale. Il a la forme

|| BINARY || ou || DECIMAL ||

respectivement.

EXEMPLE 9.23. L'instruction || DECLARE ALPHA DECIMAL, X2 BINARY ; || déclare décimale la variable ALPHA et binaire la variable X2 (voir le p. 9.7.17).

Le *descripteur de forme de représentation des nombres* indique si les valeurs d'une variable sont en virgule fixe ou en virgule flottante. Il a la forme respective

|| FIXED || et || FLOAT ||.

EXEMPLE 9.24. L'instruction || DECLARE A FLOAT, B FIXED ; || déclare que les valeurs de la variable A sont flottantes et celles de la variable B fixes.

Le *descripteur de type* indique qu'une variable est réelle, complexe ou logique. Il a la forme respective

|| REAL || COMPLEX || LOGICAL ||.

EXEMPLE 9.25. L'instruction || DECLARE A REAL, B COMPLEX; || déclare la variable A réelle et la variable B complexe.

Le *descripteur de précision* indique le nombre de chiffres significatifs, binaires ou décimaux, qu'il faut conserver et le facteur d'échelle. Ce descripteur a l'une des formes suivantes

(n) ou (n, p),

où n est un entier décimal indiquant le nombre de chiffres significatifs binaires ou décimaux qu'il faut conserver dans la valeur d'une variable représentée à point fixe ou flottant ; p est ou bien vide, ou bien un facteur d'échelle (entier décimal signé ou non) qui dit où placer le point dans la suite de n chiffres. (Il définit le nombre de chiffres après le point et n'est utilisé que pour les variables à point fixe.) Si le facteur d'échelle n'est pas présent et que la variable soit à point fixe, ce facteur est considéré comme nul.

Le descripteur de précision ne s'utilise qu'avec un descripteur de type ou un descripteur de forme de représentation et suit immédiatement ces derniers.

EXEMPLE 9.26. L'instruction || DECLARE ALPHA FLOAT (3), A REAL (6) DECIMAL, B FIXED (5,2)|| déclare que la variable ALPHA est à point flottant et qu'il faut conserver dans ses valeurs trois chiffres significatifs, que la variable A est réelle à point flottant et avec six chiffres significatifs à conserver, la variable B est à point fixe et avec cinq chiffres significatifs à conserver. Pour cette dernière est indiqué, de plus, le facteur d'échelle égal à 2. Cela signifie que, si une valeur de la variable B s'obtient égale à 0.0123, il faut la mettre sous la forme 1.23.

REMARQUE 9.1. Les descripteurs énumérés ne s'utilisent pas avec le descripteur || PICTURE||.

Déclaration implicite de variables arithmétiques. Lorsque la base de système de numération, la forme de représentation et le type des valeurs d'une variable ne sont pas déclarés, ces caractéristiques sont attribuées à la variable d'après la première lettre de son identificateur, selon la règle suivante: si la première lettre est l'une des lettres || I || J || || K || L ||M|| N ||, les descripteurs implicites sont || FIXED|| REAL||BINARY;|| dans le cas contraire les descripteurs sont || FLOAT|| REAL|| DECIMAL||.

Lorsque l'un ou l'autre des descripteurs fait défaut, il sera attribué automatiquement; ce sera || DECIMAL|| pour la base de numération, || FLOAT|| pour la forme de représentation, || REAL|| pour le type.

Le *descripteur* || PICTURE|| sert à définir le format d'un nombre ou d'une chaîne de caractères.

Le descripteur || PICTURE|| pour les nombres réels a la forme

$$\|\ \text{PICTURE}\ 'x'\ \|,$$

où x est un spécificateur dépendant de la forme du nombre. Ainsi, pour un nombre binaire à point fixe, le spécificateur a la forme

$$yy_1y \dots y_iVy_{i+1} \dots y_nP, \qquad (9.1)$$

où y peut être: a) vide, si le nombre est représenté en code complémentaire (alors $y_1 = \dots = y_n = 2$) ou en code inverse (alors $y_1 = \dots = y_n = 3$), b) un symbole S, si le nombre est représenté en code direct (alors $y_1 = \dots = y_n = 1$); S détermine son signe: ($S = 1$ pour un nombre négatif ou nul, $S = 0$ dans le cas contraire);

V est vide ou un point;

P est vide ou un facteur d'échelle qui a la forme $F\ (pq)$, où p est vide ou l'un des signes $\|+\|\ -\|$, q est un entier décimal. Le facteur d'échelle dit que le point décimal ou binaire doit être déplacé de q positions vers la droite en cas de p vide ou $\|+\|$ et vers la gauche en cas de p égal à $\|-\|$.

Pour les nombres réels binaires à point flottant, le spécificateur x a la forme

$$yy_1 \ldots y_i V y_{i+1} \ldots y_n P, \qquad (9.2)$$

où V et y, y_j $(j = 1, 2, \ldots, i, i+1, \ldots, n)$ ont le même sens que dans (9.1).

Le symbole P est ou bien $KS1t_1 \ldots t_l$, si le nombre s'écrit en code direct (alors $t_1 = \ldots t_l = 1$), ou bien $K2t_1 \ldots t_l$, si le nombre s'écrit en code complémentaire (alors $t_1 = \ldots t_l = 2$), et $K3t_1 \ldots t_l$ s'il s'écrit en code inverse (alors $t_1 = \ldots = t_l = 3$). Ici le symbole K définit le format de l'exposant, et $S1t_1 \ldots t_l$; $2t_1 \ldots t_l$ et $3t_1 \ldots t_l$ définissent l'exposant même du nombre (S et le signe du nombre).

Pour les nombres décimaux à point fixe, le spécificateur a la forme

$$S9 \ldots 9V9 \ldots 9P,$$

où S, V et P ont la même signification que dans (9.1).

Pour les nombres décimaux à point flottant, le spécificateur a la forme

$$S_19 \ldots 9V9 \ldots 9TS_29 \ldots 9,$$

où S_1 est le signe du nombre, S_2 le signe de l'exposant, le symbole T coïncide avec l'un des symboles || E || K ||, $TS_29 \ldots 9$ détermine l'exposant du nombre.

On peut utiliser avec le descripteur || PICTURE || les symboles suivants: 1) || Z || (symbole de suppression de zéros) qui signifie que les zéros à la tête des nombres représentés suivant la construction donnée sont à remplacer par des blancs; 2) || * || qui signifie que ces zéros sont à remplacer par || * || (il est interdit d'utiliser les deux symboles || Z || et || * || dans un format); 3) || Y || qui signifie que, si la position donnée est occupée par un zéro, il faut le remplacer par un blanc; dans le cas contraire il faut garder le chiffre non nul (cela concerne tous les zéros et non seulement les zéros en tête de nombres).

Le nombre de chiffres d'une construction numérique à point fixe est déterminé par un couple de nombres (μ, ν) où μ est le nombre total des positions dans la construction, ν le nombre de positions après le symbole V. S'il y a un facteur d'échelle F (pq), les longueurs mentionnées sont données par le couple de nombres $(\mu, \nu - q)$.

Le *descripteur de chaînes* indique si les chaînes de données sont des suites binaires ou des chaînes de caractères. Il y a deux types de descripteurs de chaînes. Le premier est de la forme

$$y\ (z)\ x$$

où y est ou bien le descripteur || BIT || ou bien le descripteur || CHARACTER ||; z (la quantité indiquant la longueur réelle d'une

chaîne de longueur fixe et la longueur maximale d'une chaîne de longueur variable) est ou bien un entier décimal non signé (si la chaîne est déclarée avec le descripteur || STATIC||), ou bien une expression arithmétique (voir le p. 9.5.1) dont la valeur doit être calculée et transformée en un entier au moment de stockage de chaînes ou au moment d'entrée dans un bloc déclarateur (voir le p. 9.7.1), ou bien le symbole || * || (si les longueurs peuvent être prises telles que l'on a fixées, lors de l'allocation précédente, pour les paramètres ou les variables contrôlées non basées (voir le p. 9.7.14, 9.7.15), ou si la longueur doit être indiquée par l'instruction || ALLOCATE || qui suit (pour les variables commandées non basées); x est ou bien vide (si la chaîne est de longueur fixe), ou bien le descripteur || VARYING || (si la chaîne est de longueur variable).

Le deuxième type de descripteur de chaîne a la forme

|| PICTURE '*a*' ||,

où *a* est un spécificateur de format qui se compose de symboles A, X et de chiffres 9. La lettre A désigne n'importe quelle lettre ou le blanc à la position correspondante du format, la lettre X désigne n'importe quel symbole, le chiffre 9 un chiffre décimal ou un blanc.

Exemple 9.27. L'instruction || DECLARE BETA CHARACTER (6), GAMMA BIT (10), X24 PICTURE 'AXAA99AX', YZ2 BIT (*) VARYING; || déclare la variable BETA chaîne de caractères dont la longueur vaut 6 symboles, la variable GAMMA chaîne binaire dont la longueur vaut 10 bits, la variable X24 chaîne de caractères de format || 'AXAA99AX' ||, la variable YZ2 chaîne binaire dont la longueur maximale doit être prise de l'allocation précédente ou indiquée dans l'instruction || ALLOCATE || qui suit.

Le descripteur || LABEL || indique que les valeurs d'une variable sont des étiquettes d'instructions. Il a la forme

|| LABEL (n_1, n_2, . . ., n_m) ||,

où n_i ($i = 1, 2, \ldots, m$) est une étiquette d'instruction. Une variable décrite avec un descripteur || LABEL || ne peut prendre que des valeurs de la liste (n_1, . . ., n_m) si celle-ci n'est pas vide.

Le *descripteur* || TASK || indique que la variable donnée est un nom de branche.

Le descripteur de branche peut être utilisé non seulement dans une instruction || DECLARE ||, mais aussi dans un || CALL ||.

Un nom de branche peut être associé avec d'autres descripteurs, par exemple, avec ceux de dimensions, de domaines d'actions, d'allocation de mémoire, ainsi qu'avec || DEFINED || ABNORMAL || SECONDARY ||.

On peut également utiliser un nom de branche en tant qu'argument d'une fonction incorporée || PRIORITY || (voir le § 9.9) et en tant que paramètre d'une procédure.

Le *descripteur* || EVENT || indique que la variable donnée est utilisée comme nom d'événement.

En plus de l'instruction || DECLARE ||, le descripteur d'événement peut être utilisé dans les instructions || CALL || DISPLAY || || WAIT || et comme argument d'une fonction standard ||EVENT|| (voir le § 9.9).

Un nom d'événement peut être utilisé avec d'autres descripteurs: avec ceux de dimensions, de domaines d'action, d'allocation de mémoire, avec || DEFINED || ABNORMAL || SECONDARY ||.

Le *descripteur de dimensions* définit les bornes d'un tableau et a la forme

$$(a_1 : b_1,\ a_2 : b_2,\ \ldots\ a_n : b_n),$$

où a_i (la borne inférieure du i-ème tableau) est ou bien vide (alors on pose $a_i = 1$), ou bien une expression aritmétique dont la valeur doit être calculée et transformée en un entier vers le moment d'allocation de mémoire pour le tableau, ou bien un || * ||, si la borne inférieure véritable doit être prise de l'allocation précédente ou indiquée dans l'instruction || ALLOCATE || suivante, ou bien un entier décimal, si le tableau est déclaré statique; b_i ($i = 1, 2, \ldots, n$) est la borne supérieure du j-ème tableau. Cette quantité, tout comme a_i, peut être une expression, un symbole || * || ou un entier décimal, mais ne peut pas être vide.

Lorsque, dans la déclaration d'une structure, l'une quelconque des bornes est le symbole || * ||, toutes les bornes pour la structure de niveau supérieur et pour les autres éléments de la structure doivent être elles aussi les symboles || * ||. Le symbole: || * || ne peut pas être utilisé pour désigner les bornes d'un tableau (d'une structure) de variables basées (voir le § 9.3).

Exemple 9.28. L'instruction || DECLARE A1 (—2 : 6,7), K (*, *); || déclare que: a) le tableau A1 est un tableau bidimensionnel de 9 lignes et 7 colonnes; b) le tableau X est un tableau bidimensionnel dont les bornes doivent être prises de l'allocation de mémoire précédente ou établies dans l'instruction || ALLOCATE || suivante.

Le *descripteur d'allocation de mémoire extérieure* || SECONDARY || indique qu'il est possible de ranger dans la mémoire extérieure une structure de niveau supérieur, un tableau ou une variable qui ne font pas partie de quelques structures ou tableaux.

Le *descripteur* || ABNORMAL || indique que les valeurs d'une variable doivent se trouver constamment dans la mémoire principale.

Le descripteur || NORMAL || indique que la variable donnée est ordinaire et ses valeurs peuvent être stockées dans la mémoire intérieure (principale) ou extérieure (secondaire).

Les variables sans descripteurs || NORMAL || et || ABNORMAL || sont considérées comme ordinaires (normales).

Les *descripteurs* || REDUCIBLE || et || IRREDUCIBLE || servent à décrire les noms d'entrée dans les procédures (voir les pp. 9.7.8, 9.7.9.)

Il existe des procédures définitivement irréductibles, complètement irréductibles et réductibles.

Une procédure est dite *définitivement irréductible* si elle (ou l'une des procédures qu'elle appelle) utilise, modifie, range ou supprime les données extérieures, ou bien modifie, range ou supprime les arguments. De plus, une procédure intérieure est définitivement irréductible, si elle utilise, modifie, range ou efface certaines variables connues dans un bloc appelant cette procédure.

Une procédure est *complètement irréductible* si elle (ou l'une des procédures qu'elle appelle) possède l'une au moins des propriétés suivantes: a) fournit des valeurs non identiques d'une fonction pour une même valeur de l'argument; b) exécute des opérations d'entrée-sortie; c) assure la sortie de la procédure au moyen de l'instruction || GO TO ||. Une procédure complètement irréductible (ses entrées) doit être décrite par le descripteur || IRREDUCIBLE ||.

Une procédure définitivement irréductible peut être décrite elle aussi par le descripteur || IRREDUCIBLE ||.

Dès que l'irréductibilité est spécifiée l'hiérarchie des opérations devient essentielle, par exemple pour le calcul d'une expression. Il est interdit d'essayer une optimisation d'un bloc irréductible du programme en composition.

Le *descripteur* || REDUCIBLE || indique que le nom d'entrée se rapporte à une procédure qui n'est pas irréductible.

Un appel à une procédure sans descripteur est considéré comme irréductible, s'il n'est pas un appel à une fonction extérieure; dans le cas contraire il est considéré comme réductible.

Les *descripteurs* || USES || et || SETS || servent à indiquer l'irréductibilité du nom d'entrée à une procédure opérant une transformation de données.

Le descripteur || USES || a la forme

$$\|\,\text{USES}\,\|\;(x_1,\ x_2,\ \ldots,\ x_n)\,\|,$$

où x_i $(i = 1, 2, \ldots, n)$ est ou bien un entier décimal déterminant le numéro de l'argument d'appel de la procédure pour l'entrée nommée, ou bien un nom de variable accessible à la procédure appelée et au bloc contenant une déclaration avec le descripteur donné, ou bien le symbole || * || qui désigne tout l'ensemble des variables pour la procédure appelée et pour le bloc contenant une déclaration

avec le descripteur donné. Aucune valeur n'est attribuée à l'élément x_i dans la procédure appelée, ni dans les procédures qu'elle appelle, si seulement cet élément n'est pas indiqué dans le descripteur || SETS ||, et, de plus, ces procédures n'appellent aucun autre élément accessible au bloc à l'exception des données explicitées dans les arguments des instructions || DECLARE ||. des instructions avec l'option || CALL || et des appels de fonctions.

Le descripteur || SETS || a la forme

|| SETS (x_1, . . ., x_n) ||,

où x_i sont les mêmes que dans le descripteur || USES ||. Mais cette fois les éléments x_i sont tels que la procédure appelée ou les procédures qu'elle appelle ne s'occupent que d'eux (leur attribuent les valeurs, les stockent dans la mémoire ou les en enlèvent à condition que ces éléments ne soient pas utilisés dans le descripteur || USES ||), et laissent de côté toutes les autres données.

Si un élément de liste d'un descripteur de type considéré est déterminé d'après une base (voir le descripteur || CONTROLLED ||), et si la base est accessible aux blocs appelant et appelé, la base doit figurer elle aussi dans la liste.

9.6.2. Descripteurs de nom d'entrée. Un nom d'entrée peut être décrit par un descripteur || ENTRY ||. Pourtant, un nom d'entrée peut être accompagné d'autres descripteurs dont || SETS || USES || || RETURNS || BUILTIN ||. Lorsqu'on se sert de l'un de ces descripteurs pour un nom d'entrée donné, on se passe du descripteur || ENTRY ||. Un nom d'entrée peut également avoir les descripteurs || IRREDUCIBLE || REDUCIBLE || GENERIC ||. Le dernier descripteur signifie que l'identificateur donné est déclaré comme nom générique d'une famille d'entrées.

Voyons de plus près certains des descripteurs énumérés.

Le *descripteur* || ENTRY || sert à déclarer, à l'intérieur d'une procédure, le nom de ses entrées. Il a la forme

|| ENTRY (x_1, x_2, . . ., x_k) ||,

où x_i est ou bien vide, ou une liste de descripteurs du i-ème paramètre *) (la liste des descripteurs comprend tous les descripteurs définissant le paramètre donné), k est le nombre de paramètres de l'entrée décrite.

Le descripteur || ENTRY || s'utilise dans les cas où : a) une étiquette d'entrée est indispensable dans la procédure ; b) il n'y a pas d'appel du nom d'entrée donné au moyen de l'opérateur || CALL || ou d'un indicateur de fonction (si cet indicateur n'est pas déclaré avec l'un des descripteurs || SETS || USES || BUILTIN || RETURNS ||).

*) On entend par paramètre un identificateur figurant dans une instruction || PROCEDURE || ou || ENTRY ||.

EXEMPLE 9.29. L'instruction || DECLARE B ENTRY (FIXED); || déclare que l'identificateur B est une entrée dans une procédure, et que cette entrée a un paramètre déclaré comme un nombre à point fixe, décimal, réel (les deux derniers descripteurs sont définis d'une manière implicite).

Le *descripteur* || GENERIC || est utilisé pour attribuer à un identificateur la propriété d'être le nom générique d'une famille d'entrées. Il a la forme

$$|| \text{GENERIC } (y_1, \ldots, y_n) ||,$$

où y_i est un descripteur de nom d'entrée ayant la forme || ENTRY $(x_1^{(i)}, \ldots, x_{n_i}^{(i)})$ ||.

Le *descripteur* || BUILTIN || indique que l'appel de l'identificateur décrit par ce descripteur doit être considéré comme l'appel d'une fonction standard (incorporée) dont le nom coïncide avec l'identificateur donné dans le bloc contenu dans un autre bloc où ce nom a été déclaré pour d'autres raisons.

Si le descripteur || BUILTIN || est utilisé pour un nom d'entrée, ce nom peut ne plus avoir de descripteurs.

Le descripteur || BUILTIN || ne peut pas être utilisé pour les paramètres formels.

Les *descripteurs de domaine d'action* servent à indiquer des domaines d'actions où les noms déclarés sont accessibles. Ils ont la forme

|| INTERNAL || où || EXTERNAL ||.

Si un nom est déclaré dans l'algorithme, cela veut dire qu'il existe un domaine bien déterminé de l'algorithme où cette déclaration reste en vigueur. Ce domaine s'appelle *domaine d'action du nom* déclaré.

Le domaine d'action d'un nom se limite à un bloc pour lequel cette déclaration est intérieure, exception faite pour les blocs contenus dans celui-là, pour lesquels une autre déclaration du même nom est intérieure. Ce qu'on a dit concerne toutes les déclarations de noms, à l'exception de déclarations de noms d'entrée de procédures extérieures qui sont considérées comme des déclarations explicites déclarant ce nom en tant que nom d'entrée avec le descripteur || EXTERNAL ||. Le domaine d'action d'une telle déclaration est toute la procédure extérieure, à l'exception de ceux de ses blocs qui contiennent une autre déclaration du même nom considérée comme intérieure. On considère que toutes les déclarations extérieures d'un même nom dans un algorithme définissent un seul nom. Son domaine d'action sera la réunion des domaines d'action de toutes ses déclarations extérieures.

Le descripteur || RETURNS || s'utilise avec un nom d'entrée déclaré explicitement pour indiquer les descripteurs des identifi-

cateurs des variables (des tableaux, des structures) que la procédure fournit lorsqu'on l'appelle par cette entrée. Le descripteur a la forme

|| RETURNS (x)||,

où x est un bien vide, ou bien un ensemble de descripteurs qui caractérisent les données indiquées. Ces descripteurs peuvent être descripteurs de chaînes, descripteurs arithmétiques (numériques) et descripteurs de pointeurs. Si x est vide, alors les identificateurs des données fournies par la procédure se voient munis de propriétés (de descripteurs) d'après les règles de déclaration implicite (par défaut), selon le type des données.

9.6.3. Descripteurs d'allocation de mémoire. Ils servent à indiquer le mode de rangement des variables dans la mémoire.

Il existe trois modes d'allocation de mémoire : statique, automatique et contrôlée. Pour spécifier un mode déterminé, on déclare la variable correspondante avec l'un des descripteurs ||STATIC|| AUTOMATIC|| CONTROLLED|| selon le cas.

Selon le *mode d'allocation de mémoire statique* la variable est mise en mémoire avant l'exécution du programme et y conserve sa place jusqu'au moment où l'exécution de tout le programme soit terminée.

Le descripteur de domaine d'action d'une variable stockée de la manière statique peut être intérieur et extérieur. Si le domaine d'action d'une variable est extérieur, et que le descripteur d'allocation de mémoire ne soit pas donné, alors on considère la variable comme stockée dans la mémoire de manière statique.

Le *mode d'allocation de mémoire automatique* prévoit que la variable est mise en mémoire avant chaque entrée dans un bloc pour lequel la déclaration de la variable est intérieure. La mémoire est vidée à la sortie du bloc. Si le bloc représente une procédure récursive (qui s'exécute plusieurs fois avant la sortie définitive du bloc), alors, tant que la procédure s'exécute, la partie de mémoire réservée à la variable « descend » à chaque nouvelle entrée et « remonte » à chaque sortie, ce qui est analogue au fonctionnement d'une mémoire à magasin.

Le descripteur de domaine d'action d'une variable stockée automatiquement ne peut qu'être intérieur. Si le domaine d'action d'une variable est intérieur, et que le descripteur d'allocation de mémoire soit absent, alors cette variable sera rangée de façon automatique.

Selon *le mode d'allocation de mémoire contrôlé* une variable est mise en mémoire à la suite d'exécution d'une instruction || ALLOCATE || qui contient son nom ; la place en mémoire occupée par la variable est libérée à la suite d'exécution d'une instruction || FREE || contenant le nom de la variable à effacer (voir les

pp. 9.7.14, 9.7.15). A un certain moment d'exécution du programme il peut s'avérer inconnu, si une variable x est stockée ou non dans la mémoire. Le test d'allocation s'effectue alors à l'aide de la fonction incorporée || ALLOCATION (x) || (voir le § 9.9) qui sort la valeur || '1'B || si x est stockée dans la mémoire, et la valeur || '0'B || dans le cas contraire.

EXEMPLE 9.30. L'instruction || DECLARE X1 STATIC, Y CONTROLLED, Z27; || déclare que la variable X1 est statique, la variable Y est contrôlée et la variable Z27 est automatique par omission.

En plus de la méthode décrite d'allocation de mémoire contrôlée pour une variable (un tableau, une structure), il existe une autre option d'allocation contrôlée définie par un descripteur de la forme

|| CONTROLLED (x) ||,

où x est un identificateur déclaré explicitement ou implicitement comme pointeur; de toute façon, il doit être considéré comme variable du type pointeur.

Une variable (un tableau, une structure) pourvue d'un tel descripteur s'appelle *variable basée*, et la variable de nom x s'appelle base.

Le descripteur || CONTROLLED (x) || indique que la variable basée est stockée dans la mémoire spécifiée par la variable x.

EXEMPLE 9.31. L'instruction || DECLARE A POINTER, RQ FIXED CONTROLLED (A); || déclare que la variable A est un pointeur (voir le descripteur || POINTER ||) et la variable RQ est basée, à point fixe, réelle, décimale (les deux dernières propriétés sont définies par omission), stockée dans l'endroit spécifié par l'identificateur A.

EXEMPLE 9.32. L'instruction || DECLARE GAMMA FLOAT DECIMAL CONTROLLED (Z); || déclare que la variable GAMMA est basée, à point flottant, décimale, réelle (la dernière propriété est définie par omission), que sa place dans la mémoire est spécifiée par l'identificateur Z; Z est une variable pointeur par omission.

Le *descripteur de pointeur* || POINTER || indique que la variable donnée est du type pointeur et qu'on peut lui appliquer n'importe quel mode d'allocation de mémoire. Ce descripteur peut s'utiliser dans l'instruction || DECLARE ||.

EXEMPLE 9.33. L'instruction || DECLARE X POINTER, STATIC; || déclare explicitement la variable X comme pointeur statique.

On peut omettre le descripteur || POINTER || dans la description d'une variable (un tableau, une structure) si dans une instruction || DECLARE || il est utilisé avec un descripteur || CONTROLLED (x) || ou dans l'une des instructions || ALLOCATE || READ || LOCATE || avec le descripteur || SETS ||.

La valeur d'une variable du type pointeur peut être établie ou bien au moyen d'une instruction d'affectation (voir le p. 9.7.2), ou bien au moyen de l'option || SETS || dans les instructions || ALLOCATE || READ || LOCATE ||, ou enfin au moyen du descripteur || INITIAL ||.

Les descripteurs || AREA || CELL || sont eux aussi des descripteurs d'allocation de mémoire.

Le *descripteur de domaine* || AREA || définit des mémoires que l'on peut utiliser pour garder les valeurs de variables basées, il a la forme

$$\| \text{AREA } (m_1 (n_1) x_1, m_2 (n_2) x_2, \ldots, m_k (n_k) x_k) \|,$$

où x_i est l'ensemble des descripteurs déclarant les propriétés des grandeurs à stocker dans le domaine donné, n_i est le nombre de ces grandeurs, m_i étant ou bien vide, ou bien un niveau.

Exemple 9.34. L'instruction || DECLARE A AUTOMATIC AREA ((20) FLOAT, (6) CHARACTER (25), 1 (15), 2 FIXED, 2 POINTER ; || déclare que A est un domaine de mémoire à allocation automatique où l'on doit placer 20 constantes réelles, décimales à point flottant, 6 chaînes de caractères d'une longueur de 25 caractères chacune et un tableau de 15 structures dont chacune se compose de variables à point fixe et d'un pointeur.

Ainsi, en décrivant un domaine on ne donne pas les noms des grandeurs à stocker dans ce domaine. De plus, il n'est pas exigé que les données qui seront réellement rangées dans le domaine correspondent exactement aux déclarations faites dans le descripteur du domaine, du point de vue de leur quantité, de l'ordre de disposition, des descriptions. Il peut se trouver que le volume réservé de mémoire soit insuffisant ou, au contraire, ne soit utilisé que partiellement.

La valeur d'une variable du type domaine peut être établie à l'aide d'une instruction d'affectation (voir le p. 9.7.2).

Le *descripteur* || CELL || définit un identificateur Z en tant que cellule (voir le § 9.3) et a la forme

$$\| \text{CELL } n_1 x_1 y_1, n_2 x_2 y_2, \ldots, n_k x_k y_k \|,$$

où x_i est le nom de la grandeur dont les valeurs sont à stocker dans le domaine de mémoire de nom Z; n_i est un entier décimal non signé qui détermine le niveau de x_i ($n_i < n$, où n est le numéro du niveau

de Z) ; y_i est l'ensemble des descripteurs caractérisant les propriétés de la grandeur x_i.

Exemple 9.35. L'instruction || DECLARE 1A CELL 2B, 2C FIXED (S) ; || déclare que l'identificateur A est une cellule. Les déclarations || 2B || et || 2C FIXED (S) || forment la liste du descripteur || CELL ||.

Le descripteur || CELL || indique que chaque grandeur x_i déclarée dans le descripteur sera stockée en même endroit de mémoire défini par l'identificateur Z. Ainsi, conformément à la déclaration de l'exemple 9.35, les quantités B et C seront placées dans la même mémoire spécifiée par l'identificateur A. Pourtant, à chaque moment donné on peut stocker en même endroit les valeurs d'une seule grandeur x_i. Le descripteur || CELL || diffère donc du descripteur || DEFINED || (voir le p. 9.6.5) par le fait qu'il assure l'identité de la mémoire (i.e. les données de types différents occuperont la même mémoire), tandis que le descripteur || DEFINED || assure l'identité des données (i.e. on pourra appeler les mêmes données par des méthodes différentes).

L'identificateur Z d'une cellule peut avoir d'autres descripteurs placés avant ou après le mot de service || CELL ||, mais obligatoirement avant x_1. Ces autres descripteurs peuvent être : le descripteur de dimensions || NORMAL || ABNORMAL ||, les descripteurs d'allocation de mémoire, || EXTERNAL || INTERNAL || SECONDARY ||, les trois derniers n'étant utilisables que pour décrire l'identificateur d'une cellule de niveau 1. Chaque dimension associée à un identificateur de cellule se rapporte également à la grandeur x_i déclarée dans le descripteur || CELL ||.

Un identificateur de cellule peut aussi bien figurer dans les instructions || ALLOCATE || FREE ||.

Exemple 9.36. L'instruction || DECLARE 1A CELL CONTROLLED, 2B FLOAT (6), 2C FIXED (10) ; || déclare que A est une cellule. Pour cette variable l'accès à la mémoire est ouvert ou fermé par les instructions || ALLOCATE A || et || FREE A || respectivement (voir les pp. 9.7.14, 9.7.15). Dès le moment d'exécution de l'instruction || ALLOCATE A || et jusqu'au moment d'exécution de l'instruction || FREE A ||, la cellule A est accessible aux grandeurs B et C.

9.6.4. Descripteurs de tableaux et de structures de niveau supérieur. Ils s'utilisent pour les noms de structures de niveau supérieur ou les noms de tableaux qui ne font pas parties de structures. Ces descripteurs ont les formes respectives || PACKED || ALIGNED ||.

Le *descripteur* || PACKED || indique que les éléments de niveau inférieur d'une structure (d'un tableau) sont disposés dans la mémoire

de manière compacte, sans espaces, i.e. qu'il n'y a pas de mémoire non utilisée entre deux éléments.

Le *descripteur* || ALIGNED || indique que les éléments d'une structure (d'un tableau) peuvent être espacés dans la mémoire.

Lorsque, pour un nom de structure de niveau supérieur les descripteurs || ALIGNED || PACKED || ne sont pas mentionnés le descripteur || ALIGNED || est sous-entendu. Dans le cas d'un nom d'un tableau isolé, n'appartenant à aucune structure, on sous-entend le descripteur : || PACKED ||.

Exemple 9.37. L'instruction || DECLARE 1X (10) ALIGNED, B (5,7), 1A (5) PACKED ; || déclare que a) les éléments de la structure de niveau supérieur X suivent l'un l'autre immédiatement dans la mémoire ; b) les éléments de la structure de niveau supérieur A sont séparés ; c) les éléments du tabeau B sont séparés eux aussi (mention implicite).

9.6.5. Descripteurs d'équivalence. Le *descripteur* || DEFINED || indique que les valeurs d'une variable donnée (d'un tableau, d'une structure) qui se dit *définie* sont identifiées avec les valeurs d'une autre variable (tableau, structure) appelée *base* ; il a la forme

|| DEFINED xy ||,

où x est un identificateur (un nom) d'une variable base, y est ou bien vide, ou bien une liste d'indices. En tant que listes d'indices ou utilise les expressions arithmétiques qui peuvent contenir, si la variable définie est un tableau, des variables fictives désignées par || i IND ||, i étant un entier non signé ($1 \leqslant i \leqslant n$, n est la dimension du tableau défini).

L'identification des valeurs de la variable (du tableau, de la structure) définie avec celles de la base se fait de deux façons : 1) par définition (identification) par correspondance ; 2) par la définition (identification) par superposition.

Si l'on utilise la variable || i IND || dans les listes d'indices, ou si l'identificateur de base et l'identificateur défini sont des noms de tableaux d'une même dimension et, parmi les autres descripteurs décrivant les propriétés du tableau défini, le descripteur || POSITION || ne figure pas, alors il s'agit d'une *définition par correspondance*. Dans tous les autres cas, on a une *définition par superposition*.

Exemple 9.38. L'instruction || DECLARE A (20,20), B (10) DEFINED A (2 * 1 IND, 2 * 1 IND) ; || déclare que le tableau défini B et le tableau base A sont définis par correspondance de telle manière que les éléments du vecteur B sont les éléments diagonaux de la matrice A, c'est-à-dire qu'à l'élément B (1) correspond l'élément A (2, 2), à l'élement B (2) l'élément A (4,4), etc.

Lorsqu'on utilise une définition par correspondance, l'appel d'un élément de la grandeur définie est interprété comme appel de l'élément correspondant de la base.

La définition par correspondance suppose la même description des grandeurs base et définie. De plus, si la liste d'indices est vide, la correspondance est directe. Ceci n'est possible que lorsque les tableaux base et défini sont de même dimension. (L'appel d'un élément de la grandeur définie sera interprété comme appel de l'élément de la base qui a la même collection des indices.) Et lorsque l'identificateur de la base est suivi d'une liste d'indices, alors pour une définition par correspondance, l'une au moins des expressions d'indices doit contenir, comme on a déjà dit, une variable fictive de la forme *i* IND. Dans ce cas, l'élément de la base qui correspond à l'élément défini s'obtient en remplaçant chaque variable *i* IND dans la liste d'indices par la valeur entière du *i*-ème indice de l'élément défini. (Il est impossible d'appeler l'élément de la grandeur définie qui n'a pas d'élément correspondant dans la base.)

Un tableau défini avec l'utilisation de la variable *i* IND dans la liste d'indices ne peut servir d'argument.

La définition par superposition signifie que l'identification de la grandeur définie avec la base se fait en superposant les mémoires, c'est-à-dire que la grandeur définie occupe entièrement ou en partie la mémoire réservée à la base. Par conséquent la modification d'une grandeur peut impliquer la modification de l'autre.

La définition par superposition est autorisée dans le cas où la grandeur définie est soit arithmétique, soit de types étiquette, pointeur, domaine, événement, branche, soit une structure se composant de variables des types énumérés. Respectivement, la grandeur base doit être soit arithmétique (indicée ou non) avec les mêmes propriétés que la grandeur définie, soit de types étiquette (indicée non), pointeur (indicé ou non), domaine, événement (indicé ou non), branche (indicée ou non), soit une structure identique qui se compose de variables des types énumérés et qui est organisée de façon à ce que les couples d'éléments correspondants des structures soient susceptibles d'être définis par superposition. Une définition par superposition est admissible aussi dans le cas où la grandeur définie est ou bien une chaîne (binaire ou chaîne de caractères de longueur fixe), ou bien un tableau (une structure) compact de variables chaînes admissibles (de tabeaux de variables chaînes).

Respectivement, la grandeur base doit être ou bien une variable chaîne (binaire ou chaîne de caractères de longueur fixe) qui n'est pas une section de tableau, ou bien un tableau (une structure) compact de variables chaînes admissibles (de tableaux de chaînes).

Dans le cas d'une définition par superposition, un descripteur || DEFINED ||, peut être accompagné par un descripteur || POSITION ||

qui a la forme

$$\| \text{POSITION } (n) \|,$$

où n est un entier décimal non signé. Ce descripteur indique le numéro (n) de l'élément de la base avec lequel est identifié le premier élément de la grandeur définie. Autrement dit, il indique la valeur du déplacement par rapport au premier élément de la base dans l'identification des éléments de la base et de la grandeur définie.

Lorsque le descripteur || POSITION || n'est pas mentionné, la grandeur définie est censée d'avoir le descripteur || POSITION (1) ||.

Exemple 9.39. L'instruction || DECLARE A CHARACTER (40), B CHARACTER (10) DEFINED A, C CHARACTER (20) DEFINED A POSITION (21), D CHARACTER (10) DEFINED A POSITION (11); || déclare que la grandeur B coïncide avec les 10 premiers symboles de la grandeur A, que la grandeur C est une chaîne se composant des symboles du 21-e à 40-e de la grandeur A, que la grandeur D se compose des symboles du 11-e à 20-e de A.

Le descripteur || DEFINED || ne s'utilise pas avec les descripteurs || INITIAL || VARYING ||, d'allocation de mémoire, de domaine d'action. Le descripteur || VARYING || ne peut pas décrire un identificateur de base.

Le domaine d'action de cet identificateur se limite au bloc dans lequel est définie l'instruction || DECLARE || contenant la grandeur définie par l'identificateur donné, avec le descripteur || DEFINED ||. D'autre part, une base peut avoir un descripteur : || EXTERNAL ||. Par conséquent, son nom est accessible à tous les blocs où elle est déclarée (comme extérieure), mais le nom de la grandeur définie ne sera pas accessible. Pourtant, la valeur de la grandeur définie sera modifiée dès que se modifie la valeur de la base dans un bloc extérieur.

L'identificateur de base doit être constamment connu dans un bloc où la grandeur définie se déclare ; cet identificateur ne doit pas avoir de descripteur || DEFINED ||, il ne peut pas être une variable basée.

9.6.6. Descripteurs de données initiales. Ils servent ou bien à attribuer les valeurs initiales aux variables (tableaux, structures) au moment d'allocation de mémoire, ou bien à indiquer la procédure que l'on appelle pour exécuter l'attribution des valeurs initiales aux variables (tableaux, structures) toujours au moment de leur stockage dans la mémoire.

Dans le premier cas le descripteur a la forme

$$\| \text{INITIAL } (x_1, x_2, \ldots, x_n) \|,$$

où x_i est ou bien un nombre, ou bien une chaîne, ou bien le symbole || * ||. Si certains x_i sont égaux entre eux, et qu'ils suivent immédiatement l'un l'autre dans la liste $(x_1, \ldots, x_n)$, alors on peut abréger la notation en utilisant le facteur de répétition (k) qui dit que le nombre donné est répété k fois dans la liste.

EXEMPLE 9.40. L'instruction || DECLARE A (20) INITIAL (0, 0, 0, 0, 0, 3, 3, 3, 3, 3, 3, 5, 5, 5, —4, 5, 5, 5, —4, 7); || déclare que le tableau A se compose de 20 éléments réels décimaux à point flottant, dont les cinq premiers ont la valeur initiale, 0, les six suivants les valeurs initiales 3, les éléments du douzième au dix-neuvième ont les valeurs initiales respectives 5, 5, 5, —4, 5, 5, 5, —4, le vingtième élément a la valeur initiale 7.

Cette instruction peut s'écrire autrement, comme || DECLARE A (20) INITIAL ((5) 0, (6) 3, (2) ((3) 5, —4, 7); ||.

Si un tableau a plus d'une dimension, alors les valeurs initiales de ses éléments sont citées consécutivement, ligne après ligne (le dernier élément variant le plus rapidement).

Un facteur de répétition peut être une expression arithmétique dont la valeur doit être calculée vers le moment d'affectation des valeurs initiales aux éléments, et sa partie entière doit être dégagée.

Le facteur de répétition négatif ou nul est sans effet.

Le symbole || * || est employé dans le cas où l'on n'attribue aucune valeur initiale à l'élément correspondant.

Dans le deuxième cas le descripteur initial à la forme

$$|| \text{INITIAL CALL } x\,(y_1, \ldots, y_m) ||,$$

où x est un nom d'entrée, y_i un argument.

EXEMPLE 9.41. L'instruction || DECLARE X (10, 10) INITIAL CALL AZ (B, C); || déclare que AZ est le nom de la procédure qui attribue les valeurs initiales au tableau X de dimensions 10×10, et que B et C sont les arguments dont la procédure AZ a besoin.

On ne peut pas utiliser le descripteur initial pour les noms d'entrée, les noms de fichiers, les données avec le descripteur || DEFINED ||, les structures, les paramètres, les données de types branche, événement, domaine.

9.6.7. Descripteur d'identification de structures || LIKE ||. Ce descripteur indique que le nom déclaré est une structure qui se compose des éléments dont les noms et les descripteurs sont identiques aux noms et descripteurs des éléments d'une structure qui suit le descripteur || LIKE || et que la composition de ces structures est la même.

Le niveau de l'élément qui suit immédiatement le descripteur || LIKE || doit être inférieur ou égal au niveau de l'élément déclaré avec ce descripteur.

EXEMPLE 9.42. L'instruction || DECLARE

```
1A,
    2B1 FIXED,
        3C1 CHARACTER (5),
        3C2 FLOAT,
    2D BIT (10),
1X,
    2B1,
        3E (10) LIKE A.B1,
        3Y,
    2B2; ||
```

est équivalente à l'instruction

```
|| DECLARE
    1A,
        2B1, FIXED,
            3C1 CHARACTER (5),
            3C2 FLOAT,
        2D BIT (10)
    1X,
        2B1
            3E (10),
                4C1 CHARACTER (5),
                4C2 FLOAT,
            3Y,
        2B2; ||.
```

9.6.8. Descripteurs de fichiers. Les descripteurs de fichiers servent à décrire les propriétés des fichiers de données.

Les déclarations d'un même fichier dans plusieurs procédures extérieures ne doivent pas être contradictoires.

Les descripteurs de fichiers sont || FILE || STREAM || RECORD || || INPUT || OUTPUT || UPDATE || PRINT || SEQUENTIAL || DIRECT || BUFFERED || UNBUFFERED || BACKWARDS || EXCLUSIVE ||.

Le *descripteur* || FILE || définit un identificateur donné en tant que nom de fichier. Ce descripteur s'utilise avec tous les descripteurs énumérés. Il est donc permis d'omettre le descripteur || FILE || si l'un au moins des descripteurs de fichiers est employé ; on dit que le descripteur || FILE || est présent implicitement.

Le *descripter* || STREAM || indique que le fichier donné doit être considéré comme une suite de symboles. Ce descripteur ne s'utilise pas avec les descripteurs || RECORD || UPDATE || DIRECT || SEQUENTIAL || BUFFERED || UNBUFFERED || BACKWARDS || || EXCLUSIVE ||. Un fichier décrit avec un || STREAM || peut être employé dans les instructions || OPEN || CLOSE || GET || PUT ||.

Le *descripteur* || RECORD || indique que le fichier donné doit être considéré comme une suite d'enregistrements (de zones) de données. Un fichier avec ce descripteur ne peut s'utiliser que dans les instructions || OPEN || CLOSE || READ || WRITE || REWRITE || || LOCATE || DELETE || UNLOCK ||.

Le *descripteur* || INPUT || indique que le contenu d'un fichier sera transféré dans la mémoire. Ce descripteur ne peut pas être utilisé avec les descripteurs || EXCLUSIVE || PRINT ||.

Le *descripteur* || OUTPUT || indique que les données seront extraites de la mémoire pour être enregistrées dans le fichier. Ce descripteur ne peut pas être utilisé avec les descripteurs || EXCLUSIVE || || BACKWARDS ||.

Le *descripteur* || UPDATE || indique que le fichier portant le nom déclaré peut être utilisé en entrée et en sortie. Ce descripteur ne s'emploie pas avec les descripteurs || STREAM || BACKWARDS || || PRINT ||.

Le *descripteur* || PRINT || indique que les données sont à imprimer. Il ne s'utilise pas avec le descripteur || RECORD ||.

Dans la description d'un fichier décrit avec un || PRINT || sont implicitement présents les descripteurs || OUTPUT || STREAM ||.

Les *descripteurs* || DIRECT || SEQUENTIAL || s'utilisent avec le descripteur || RECORD || et indiquent la manière d'accéder aux enregistrements du fichier. Dans le premier cas l'entrée-sortie des enregistrements se fait d'après un mot-clé (voir le p. 9.7.18.10 et le descripteur || KEY ||), donc le descripteur || DIRECT || s'utilise toujours avec le descripteur || KEY || ; dans le deuxième cas l'entrée-sortie se fait dans l'ordre de succession des enregistrements dans le fichier.

Les *descripteurs* || BUFFERED || UNBUFFERED || ne s'utilisent qu'avec les descripteurs || SEQUENTIAL || RECORD || et indiquent s'il faut ou non se servir de la mémoire intermédiaire (tampon) au cours des opérations d'entrée-sortie.

Si un tampon doit être utilisé, on peut le faire accessible en lui associant une variable du type pointeur qui sera utilisée pour définir une variable basée décrivant l'enregistrement sur le tampon.

Le *descripteur* || BACKWARDS || s'utilise avec les descripteurs || SEQUENTIAL || INPUT || et indique que l'entrée des données provenant d'un fichier se fait dans l'ordre inverse.

Le *descripteur* || EXCLUSIVE || s'emploie avec les descripteurs || DIRECT || UPDATE || et interdit la lecture, l'effacement ou le réenregistrement d'un article de fichier dans la branche donnée, pour la raison que dans une autre branche, cet article est en train d'être lu, effacé ou réenregistré.

Le *descripteur* || KEY || a la forme

|| KEY *n* ||,

où *n* est un entier décimal non signé qui détermine la longueur du mot-clé (voir le p. 9.7.18). Ce descripteur associe à chaque enregistrement d'un fichier un mot-clé permettant l'accès direct, ou sélectif, aux enregistrements de ce fichier.

Le *descripteur* || KEY || s'utilise avec les descripteurs || STREAM || || PRINT ||.

§ 9.7. Instructions

Les principaux éléments de structure du langage PL/1 sont les instructions.

Les instructions de PL/1 sont classées en deux grandes catégories : en instructions exécutables et non exécutables.

Les *instructions non exécutables* sont || FORMAT || DECLARE || || ENTRY ||. Elles décrivent certaines données et sont sautées lorsque le contrôle passe à elles au cours de l'exécution du programme. Toutes les autres *instructions* sont *exécutables.*

A l'intérieur d'un bloc, les instructions s'exécutent dans l'ordre de leur succession. L'ordre d'exécution séquentiel est modifié: par l'instruction de retour || RETURN ||, par l'instruction d'appel d'une procédure || CALL ||, par l'instruction de fin || END ||, par l'instruction || SIGNAL || imitant la réaction à une interruption, par l'instruction de passage de contrôle || GO TO ||, par l'instruction de procédure ||PROCEDURE ||, par l'instruction d'arrêt || STOP ||, par l'instruction « si » || IF ||.

Si le contrôle passe à une instruction non exécutable, il est transmis à l'instruction qui suit immédiatement cette instruction non exécutable.

On peut classer les instructions exécutables en les groupes suivants: les instructions de contrôle (|| GO TO || IF || DO || CALL || || RETURN || WAIT || STOP || DELAY ||) ; les instructions de structure du programme (|| PROCEDURE || || BEGIN || END || DO || || ENTRY ||) ; les instructions d'affectation, les instructions d'allocation de mémoire (|| ALLOCATE ||, les instructions d'entrée-sortie).

Une instruction du langage PL/1 a la forme

|| yxQ ; ||,

où y est ou bien vide, ou bien une construction de la forme || (y_1, y_2, ..., y_n) : || qui spécifie la situation d'interruption et où y_i est un nom de situation coïncidant avec l'un des mots réservés *) || UNDERFLOW || OVERFLOW || ZERODIVIDE|| FIXEDOVERFLOW || CONVERSION || SIZE || SUBSCRIPTRANGE || NOUNDERFLOW || NOOVERFLOW || NOZERODIVIDE || NOFIXEDOVERFLOW || NOCONVERSION || NOSIZE || NOSUBSCRIPTRANGE || ENDPAGE || ENDFILE || UNDEFINEDFILE || ou avec l'une des constructions || CHECK (z_1, z_2, ..., z_k) || NOCHECK || (z_1, ..., z_k) || dans lesquelles z_j (j = 1, 2, ..., k) est un identificateur; x est ou bien vide, ou bien une construction de la forme || x_1 : x_2 : ... ... : x_p : || dans laquelle x_q (q = 1, 2, ..., p) est un identificateur appelé étiquette de l'instruction; le symbole || ; || indique la fin de l'instruction.

Un nom de situation spécifie la nature de la situation qui doit ou ne doit pas causer une interruption, si cette situation se produit à la suite d'exécution d'une instruction qui contient son nom, sauf les instructions || PROCEDURE || BEGIN || IF || ON ||, c'est-à-dire que le domaine d'action d'une situation d'interruption se limite à l'instruction contenant son nom.

Si le nom d'une situation est contenu dans l'une des instructions || PROCEDURE || BEGIN ||, alors le domaine d'action de cette spécification sera tout le bloc (voir le p. 9.7.1) ou toute la procédure (voir les pp. 9.7.8, 9.7.9.) définis par cette instruction, y compris tous les blocs emboîtés, à l'exception des blocs et des instructions pour lesquels l'interruption est redéfinie.

Si le nom de situation est spécifié dans l'une des instructions || IF || ON ||, alors le domaine d'action de la spécification ne contient pas les blocs ni les boucles qui en font partie.

La réaction à une interruption peut être soit prédéfinie (voir le p. 9.8.1), soit prévue par le programmeur. Celui-ci définit la réaction à l'interruption dans l'instruction || ON || (voir le p. 9.8.2) ou dans l'instruction || SIGNAL || (voir le p. 9.8.4), un mot réservé || SNAP || étant mis avant l'instruction ou le bloc qui exécute la « réaction à l'interruption ».

Voyons de plus près chacune des instructions du langage.

---

*) Ces mots signifient respectivement: disparition, dépassement de capacité, division par le zéro, dépassement de capacité fixe, conversion, dimension, intervalle de variation d'indice, sans attention à la disparition, sans attention au dépassement de capacité, sans attention à la division par le zéro, sans attention au dépassement de capacité fixe, sans attention à la conversion, sans attention à la dimension, sans attention à l'intervalle de variation d'indice, fin de page, fin de fichier, fichier indéfini, vérification, sans attention à la vérification.

9.7.1. Bloc. Un bloc a la forme

$$\| y x_1 : x_2 : \ldots : x_p : \text{BEGIN } b_1 ; b_2 ; \ldots ; b_m ; s_1 ; s_2 ; \ldots ; \ldots ; s_n \text{ END} z ; \|,$$

où b_i sont des descriptions (des instructions ‖ DECLARE ‖); s_i des instructions quelconques; z une étiquette qui coïncide avec l'une des étiquettes x_j ($j = 1, 2, \ldots, p$).

Les variables décrites dans un bloc sont dites locales par rapport à ce bloc, et les variables qui ont un sens en dehors du bloc sont dites globales.

Un bloc, de même qu'une instruction isolée, est exécuté dans l'ordre naturel de succession des instructions, il peut se trouver partout où l'on peut placer une instruction isolée. Tout bloc peut contenir un autre bloc, mais aucuns deux blocs ne peuvent avoir de partie commune. Pourtant, aucun programme dans le langage PL/1 ne peut se composer d'un seul bloc.

L'exécution d'un bloc commence par ce que toutes les variables qui y sont décrites acquièrent un nouveau sens, conformément aux descriptions données dans le bloc. Puis sont exécutées les instructions du bloc dans l'ordre de leur notation (à moins que l'exécution de certaines d'entre elles ne consiste en la modification de l'ordre d'exécution). Avant la « sortie » du bloc, la mémoire occupée par les variables locales est vidée.

9.7.2. Instructions d'affectation. L'*instruction d'affectation* a la forme

$$\| v_1, v_2, \ldots, v_n = wy ; \|.$$

Les significations des w, y et v_i peuvent être les suivantes:

1. w est une expression scalaire, y est vide, v_i ($i = 1, 2, \ldots, n$) est soit un nom de variable scalaire, soit un nom de pseudo-variable, soit un nom de tableau, soit enfin un nom de pseudo-tableau;

2. w est une expression du type tableaux (structures ou pseudo-structures), y est soit vide, soit une construction ‖, BYNAME ‖, v_i ($i = 1, 2, \ldots, n$) est un nom de tableau ou de pseudo-tableau (un nom de structure ou de pseudo-structure);

3. w est soit une étiquette, soit un nom de variable du type étiquette, y est vide, v_i ($i = 1, 2, \ldots, n$) est un nom de variable du type étiquette ou un tableau du type étiquette;

4. w est une expression du type pointeurs ou un nom de tableau du type pointeur, y est vide, v_i ($i = 1, 2, \ldots, n$) est un nom de variable du type pointeur ou d'un tableau du type pointeur.

On entend par pseudo-variable l'une des fonctions incorporées suivantes: ‖ COMPLEX (a, b) ‖ REAL (c) ‖ IMAG (c) ‖ SUBSTR (s, i, j) ‖ EVENT (a) ‖ PRIORITY (a) ‖ UNSPEC (a) ‖ (voir le § 9.9) dont les arguments sont des variables de type admissible;

on entend par pseudo-tableau une pseudo-variable dont les arguments sont des noms de tableaux.

L'exécution d'une instruction d'affectation commence par le calcul de toutes les expressions se trouvant à gauche du signe d'égalité (dans les indices ou dans les arguments de pseudo-variables). Ces expressions sont calculées dans l'ordre de leur succession de la gauche vers la droite, une seule fois chacune. Puis on calcule l'expression dans la partie droite, si l'instruction d'affectation est de première espèce et que sa partie gauche soit une variable.

S'il le faut, la valeur d'une expression est transformée selon les spécifications des variables v_i (voir les pp. 9.5.1., 9.5.3).

Dans une instruction d'affectation de première espèce, les variables dont les noms sont indiqués dans la partie gauche peuvent être des variables arithmétiques.

Exemple 9.43. Soient A, B et C des variables arithmétiques. Alors l'instruction || A, B, C = A ** 2 + B/A + C|| est une instruction d'affectation de première espèce. Elle est équivalente aux trois instructions suivantes: || A = X ; || B = X ; || C = X; ||, où || X= = A**2 + B/A + C ; ||, X et l'expression A**2 + B/A + C ayant les mêmes spécifications.

Si v_i est une variable dont la valeur est une chaîne de longueur fixe et inférieure à la longueur de w, alors la chaîne qui représente la valeur de w est tronquée ; si c'est la longueur de w qui est inférieure, on la complète à droite par des zéros (pour une chaîne binaire) ou par des espaces (pour une chaîne de caractères).

Si la longueur de la chaîne v_i est variable, et que w dépasse la longueur maximale de la chaîne v_i, la valeur de w est tronquée et rendue égale à la longueur maximale; si la valeur de w est plus courte que la longueur maximale de la chaîne, une affectation sera effectuée et la longueur de w sera considérée comme une nouvelle longueur de la chaîne.

Si v_i est un nom de tableau ou de pseudo-tableau (dans ce cas tous les tableaux v_i doivent avoir les mêmes dimensions et les mêmes bornes), alors la valeur de w est attribuée à tous les éléments du tableau v_i. Ce processus est équivalent au processus suivant.

Supposons que l'instruction d'affectation ait la forme

|| L1 : A, B = ⟨expression⟩ ; ||,

où A et B sont des tableaux de dimension n.

Les affectations seront faites dans l'ordre suivant d'instructions:

```
|| L1 : DOI1 = L BOUND (A, 1) TO HBOUND (A, 1);
        DOI2 = L BOUND (A, 2) TO HBOUND (A, 2);
        . . . . . . . . . . . . . . . . . . . . . . . .
```

DOIn = L BOUND (A, n) TO HBOUND (A, n);
A (I1, I2, . . ., In), B (I1, I2, . . ., In) =
= ⟨expression⟩; ||.

Ainsi, une affectation de la valeur d'une expression à des tableaux se réduit à une suite d'affectations scalaires. Ici LBOUND (A, i) et HBOUND (A, i) sont les fonctions incorporées (voir le § 9.9), || DO || est l'instruction de boucle (voir le p. 9.7.5).

Dans le deuxième cas, l'instruction d'affectation est considérée comme une suite d'instructions d'affectation scalaires pour les éléments d'un tableau ou d'une structure, si y est vide. Tous les tableaux v_i et tous les tableaux dans w doivent avoir les mêmes dimensions et les mêmes bornes.

Exemple 9.44. Supposons que le tableau A représente la matrice $\begin{pmatrix} 2, & 1 \\ 4, & 5 \end{pmatrix}$, le tableàu B, la matrice $\begin{pmatrix} 7, & 3 \\ 4, & -1 \end{pmatrix}$, et l'instruction d'affectation ait la forme || X = (A + B) * 2 — A (2, 2); ||. A la suite d'exécution de cette instruction les éléments du tableau X prendront les valeurs $\begin{pmatrix} 13, & 3 \\ 11, & 3 \end{pmatrix}$.

Si w contient des opérandes-structures, alors tous les tableaux cités dans une instruction d'affectation doivent être tableaux de srtuctures, et toutes les structures doivent être composées de la même façon (seul le nombre de niveaux peut être différent).

Exemple 9.45. Considérons la structure

|| 1A,
 2B,
 2C,
  3D,
 2E ||.

Alors l'instruction d'affectation || A = A + B; || sera développée en les instructions || B = B + B; C = C + B; || ou, définitivement en les instructions || B = B + B; D = D + B; E = = E + B; ||.

Exemple 9.46. Considérons les structures

|| 1X,		|| 1A,
2Y,		2B,
2Z,	et	2C,
3Q,		3D,
3R ||		3E ||.

Alors l'instruction d'affectation || X = X/A + A;|| signifiera au fait la suite d'instructions

|| Y = Y/B + B; Z = Z/C + C;||

ou, définitivement

|| Y = Y/B + B; Q = Q/D + D; R = R/E + E;||.

Si, dans une instruction d'affectation, y est l'option ||,BYNAME||, alors les tableaux v_i sont des tableaux de structures. Dans ce cas l'exécution de l'instruction d'affectation se réduit aux actions suivantes:

a) On prend tous les éléments de niveau supérieur de chaque structure de l'instruction.

b) On forme l'ensemble des éléments appartenant à toutes les structures de l'instruction.

c) Pour chaque élément de l'ensemble mentionné, on construit l'instruction d'affectation correspondante. L'ordre de notation de ces instructions correspond à l'ordre dans lequel les éléments choisis sont notés dans la première structure à gauche v_1.

d) Si tous les éléments qui correspondent à un élément de l'ensemble construit sont des structures ou des tableaux de structures, on forme une instruction d'affectation avec l'option || ,BYNAME||. A la base de cette instruction on construit ensuite de nouvelles instructions d'après les règles décrites plus haut. Ceci est fait pour chaque élément de l'ensemble choisi.

e) Si, parmi les éléments correspondant à un élément de l'ensemble, il n'y a pas de structures ni tableaux de structures, alors on construit une instruction d'affectation sans option || ,BYNAME||. Aucune autre instruction d'affectation ne sera construite à partir de cette instruction de base.

Exemple 9.47. Considérons les structures

```
||1A,          ||1B,          ||1C,
    2X,            2X,            3X,
     3S1,           3S1,           4L,
     3S2,           3S1D,          4S2,
     3S3,           3M,            4S1,
    2X1,           2X1,           3X1,
       3S1D,        3S3,             4L,
       3S2F,        3S2,             4S2,
       3S2G,        3S2G||,          4H3||.
    2X2,
       3L,
       3M||,
```

Conformément à la règle a) on compose tous les éléments du niveau suivant :

A.X, B.X, C.X, A.X1, B.X1, C.X1, A.X2.

Conformément à la règle b) l'ensemble des éléments communs à ces structures sera formé par X et X1.
Dans ce cas, pour l'instruction d'affectation

$$\| A = B + 2 * C; \|$$

les instructions d'affectation suivantes seront formées :

|| A.X = B.X + 2 * C.X, BYNAME;
A.X1 = B.X1 + 2 * C.X1, BYNAME; ||.

A partir de ces instructions d'affectation on peut en construire d'autres, jusqu'à atteindre le niveau inférieur.

Exemple 9.48. Soient A un tableau de structures, R une structure :

```
1A (2, 2),          || R,
      2B,               3S,
      2C,               3T,
      2D,               3U,
        3E,                SV,
        3F ||              SW ||.
```

L'instruction d'affectation || A = R; || est équivalente aux quatre instructions d'affectation suivantes :

|| A (1,1) = R; A (1,2) = R; A (2,1) = R; A (2,2) = R; ||

qui, à leur tour, sont équivalentes aux instructions :

|| A(1,1).B=S; A(1,2).B=S; A(2,1).B=S; A(2,2).B=S;
A(1,1).C=T; A(1,2).C=T; A(2,1).C=T; A(2,2).C=T;
A(1,1).E=V; A(1,2).E=V; A(2,1).E=V; A(2,2).E=V;
A(1,1).F=W; A(1,2).F=W; A(2,1).F=W; A(2,2).F=W; ||.

9.7.3. Instruction de branchement inconditionnel. Cette instruction a la forme

|| GO TO x; ||,

où x est soit une étiquette, soit une variable scalaire du type étiquette.

L'exécution de l'instruction de branchement inconditionnel se réduit au passage du contrôle à l'instruction étiquetée x, si x est une étiquette, ou à l'instruction dont l'étiquette coïncide avec la valeur de x, si x est une variable du type étiquette. Dans ce dernier cas, la valeur de x doit être calculée vers le moment d'exécution de l'instruction de branchement inconditionnel.

9.7.4. Instruction IF. L'*instruction de branchement conditionnel* a la forme

$$\| \text{IF } x \text{ THEN } yz \|,$$

où x est une expression scalaire; y est ou bien un bloc, ou bien une boucle, ou bien une instruction; z est ou bien la construction || ELSE t||, dans laquelle t est soit un bloc, soit une boucle, soit une instruction quelconque.

L'exécution de l'instruction se réduit au calcul de la valeur de l'expression x et à sa conversion en une chaîne binaire (voir le p. 9.5.3). Si un bit au moins dans cette chaîne a la valeur 1, le contrôle passe à y. Après l'exécution de y (si elle ne modifie pas l'ordre d'exécution), le contrôle revient à l'instruction qui suit immédiatement l'instruction donnée. Si tous les chiffres de la chaîne binaire ont la valeur 0, le contrôle passe à t, lorsque z est non vide, ou à l'instruction qui suit immédiatement l'instruction donnée, dans le cas contraire.

Après l'exécution de t, le contrôle est rendu à l'instruction qui le suit, si t ne modifie pas l'ordre d'exécution.

Exemple 9.49. Soit: || X1:X2:IF A ⩽ B THEN A = 0; A = = B + 1;||. Ici || X1:X2:IF A ⩽ B THEN A = 0;|| est une instruction de branchement conditionnel qui rend le contrôle à l'instruction || A = 0;|| si la condition A ⩽ B est remplie, et à l'instruction || A = B + 1;|| dans le cas contraire.

Exemple 9.50. Soit: || X1:X2:IF C = A THEN GO TO M1; ELSE IF X > Y THEN X = 0; ELSE X = Y + 4; X = X + + Y ** 2;||, où || X1:X2:IF C = A THEN GO TO M1; ELSE IF X > Y THEN X = 0; ELSE X = Y + 4;|| est une instruction de branchement conditionnel. A son tour || IF X > Y THEN X = 0; ELSE X = Y + 4;|| est encore une instruction de branchement conditionnel.

9.7.5. Instruction de boucle. L'*instruction de boucle* a la forme suivante:

$$\| \text{DO } ts \|,$$

où t est soit vide, soit l'une des constructions || $y = x_1$ TO x_2 BY x_3;|| ou || $y = x_1$ BY x_3 TO x_2;||, dans laquelle y est la variable de contrôle; x_1, x_2 et x_3 sont des expressions scalaires; s est ou bien vide, ou bien une construction || WHILE (x);|| où x est une expression scalaire.

Si t est vide et que s soit || WHILE (x);||, alors l'instruction de boucle a la forme

|| DO WHILE (x);||.

A la suite d'exécution de cette instruction, la valeur de l'expression x est calculée et convertie, s'il le faut, en une chaîne binaire. Lorsqu'un bit au moins de cette chaîne vaut 1, le programme exécute la suite d'instructions $Q_1, Q_2, \ldots, Q_n$ (on l'appelle *boucle*) qui suivent immédiatement l'instruction donnée. Aucune des instructions Q_i $(i = 1, 2, \ldots, n)$ n'est l'instruction || END||, et l'instruction Q_{n+1} est l'instruction || END||. Cela se répète jusqu'à ce que chaque bit de la chaîne représentant la valeur transformée de l'expression x ne soit zéro. Dans ce dernier cas le contrôle passe à l'instruction qui suit immédiatement l'instruction Q_{n+1}.

L'exécution des instructions

|| A:DO WHILE (x);
⟨suite d'instructions $Q_1, \ldots, Q_n$⟩
END;
B: ⟨instruction Q_{n+2}⟩||

est équivalente à l'exécution des instructions

|| A:IF x THEN; ELSE GO TO B;
⟨suite d'instructions $Q_1, \ldots, Q_n$⟩
GO TO A;
B: ⟨instruction Q_{n+2}⟩||.

Si s est vide et t est la construction || $y = x_1$ TO x_2 BY x_3;|| (ou || $y = x_1$ BY x_3 TO x_2;||), alors l'instruction de boucle a la forme

|| DO $y = x_1$ TO x_2 BY x_3;||
(ou || DO $y = x_1$ BY x_3 TO x_2;||)

qui détermine elle aussi l'ordre d'exécution des instructions formant la boucle correspondante. Cela se fait de la manière suivante: une valeur initiale (celle de l'expression x_1) est attribuée au paramètre de boucle (à la variable y), et l'on vérifie si la condition $(x_3 >= 0)$ & $(y > x_2)$ | $(x_3 < 0)$ & $(y < x_2)$ a lieu. Si elle n'est pas satisfaite, le contrôle passe à l'instruction initiale de la boucle

Q_1. Après l'exécution de l'instruction finale de la boucle, la valeur de l'expression x_3 est ajoutée à celle du paramètre y. Le processus s'itère. Dès que la condition $(x_3 >= 0)$ & $(y > x_2)$ est satisfaite, le contrôle passe à l'instruction qui suit immédiatement l'instruction Q_{n+1}. Vers le moment d'exécution de l'en-tête de boucle, les valeurs des expressions x_1, x_2, x_3 doivent être calculées. Si l'expression x_3 (incrément de boucle) est une constante égale à l'unité, alors on peut l'omettre dans l'instruction de boucle.

Enfin, si s est la construction || WHILE (x); || et t, la construction || $y = x_1$ TO x_2 BY x_3; || (ou || $y = x_1$ BY x_3 TO x_2; ||), alors, comme dans le cas précédent, l'instruction

|| DO $y = x_1$ TO x_2 BY x_3 WHILE (x); ||

détermine l'exécution des instructions formant la boucle, en assurant les répétitions de groupes d'instructions s'il le faut. La différence du cas précédent consiste en ce que, lorsque la condition $(x_3 >= 0)$ & $(y > x_2)$ | $(x_3 < 0)$ & $(y < x_2)$ n'est pas satisfaite, on vérifie si la chaîne binaire qui représente la valeur de l'expression x est une chaîne de zéros. Si cette condition supplémentaire est satisfaite, le contrôle passe à l'instruction qui suit immédiatement l'instruction Q_{n+1}, dans le cas contraire à l'instruction initiale de la boucle, i.e. à Q_1.

Il est clair que si y est une variable indicée, alors les valeurs des expressions d'indices de cette variable doivent être calculées vers le moment d'exécution de l'instruction de boucle.

Au cours de l'exécution des instructions formant la boucle, la valeur du paramètre de boucle ne doit être modifiée par aucune des instructions Q_i $(i = 1, 2, \ldots, n)$.

La sortie de la boucle n'est permise qu'à l'aide de l'instruction || GO TO ||.

Le branchement de l'extérieur vers l'une des instructions Q_i $(i = 1, 2, \ldots, n)$ n'est possible que dans le cas où t et s sont vides.

Des boucles emboîtées sont admissibles.

Exemple 9.51. Les instructions

```
|| DO I = 1 TO 10; DO J = 1 TO 15;
   SUM = 0; DO K = 1 TO 12;
   SUM = SUM + A (I, K) * B (K, J);
   END; C (I, J) = SUM; END; ||
```

permettent de calculer la matrice C égale au produit des matrices A et B (A est le nom du tableau A (10,12), B celui du tableau B (12,15)).

9.7.6. Instruction vide. L'*instruction vide* a la forme

|| ; ||.

Son exécution représente le passage du contrôle à l'instruction suivante.

9.7.7. Instruction d'arrêt. L'*instruction d'arret* a la forme

|| STOP ; ||.

Elle provoque l'arrêt d'exécution de la branche principale et de toutes les branches subordonnées (voir le § 9.3).

9.7.8. Description d'une procédure. Une *procédure* a la forme

$$|| \, yx_1 : x_2 : \ldots : x_p Q; \; b_1; \; \ldots; \; b_m; \; s_1; \; \ldots; \; s_n; \; \text{END} \; z; ||,$$

où x_i $(i = 1, 2, \ldots, p)$ est l'étiquette de l'instruction Q qui est appelée nom de l'entrée principale en procédure ; y, b_j $(j = 1, 2, \ldots, m)$, s_k $(k = 1, 2, \ldots, n)$, z ont la même valeur que pour un bloc (voir le p. 9.7.1) ; Q est l'en-tête de la procédure qu'on appelle instruction de procédure.

9.7.9. Instruction de procédure. Programme PL/1. L instruction de procédure a la forme

$$|| \, x_1 : x_2 : \ldots : x_p \; \text{PROCEDURE} \; abcd; ||,$$

où a est soit vide, soit une liste $(A_1, \ldots, A_n)$ de paramètres formels A_i dont chacun représente ou bien un nom non composé et non indicé d'un scalaire, d'un tableau ou d'une structure, ou bien un nom d'un fichier ou d'une entrée (ces noms sont donnés par des identificateurs de niveau un) ; b est soit vide, soit une construction || OPTIONS $(t_1, \ldots, t_m)$ || dans laquelle les t_i sont des options (cette construction ne s'utilise que pour une procédure extérieure et se rapporte à toutes ses entrées) ; c est soit vide, soit le mot réservé || RECURSIVE || qui décrit la procédure donnée et signifie qu'elle peut être appelée de manière récursive, i.e. que la première instruction peut recevoir la commande de l'une des instructions intérieures ; d est soit vide, soit des descripteurs de données (voir le p. 9.6.1). Ces derniers peuvent être ou bien des descripteurs du type arithmétique, ou bien ceux du type pointeur. Ils spécifient les caractéristiques des grandeurs fournies par la procédure en question, si l'on l'a appelée comme fonction par l'entrée principale. Cela veut dire que les valeurs de grandeurs spécifiées dans l'instruction de retour (voir le p. 9.7.12) sont transformées conformément aux descripteurs de données.

Si les descripteurs de données ne sont pas explicités, ou le sont en partie, on applique les règles de déclaration implicite.

Si le nombre d'étiquettes x_i est supérieur à un, alors x_1 est interprété comme l'unique étiquette de l'instruction de procédure, et chacune des étiquettes suivantes comme une entrée indépendante ayant la même liste de paramètres et les mêmes descripteurs de données que l'en-tête de la procédure.

EXEMPLE 9.52. L'instruction || X: Y: PROCEDURE;|| est équivalente aux instructions || X: PROCEDURE; Y: ENTRY;||.

Une procédure ne peut être appelée que dans le cas d'exécution d'une instruction || CALL||, i.e. d'une instruction d'appel de procédure (voir le p. 9.7.11), ou bien d'une instruction contenant l'option || CALL||, ou bien en utilisant l'appel d'une fonction au moyen d'un pointeur de fonction (voir le § 9.4).

Une fois la procédure appelée, les instructions qu'elle contient s'exécutent. Lorsque le contrôle passe à une instruction || END||, l'exécution de la procédure s'arrête, et l'exécution du programme se poursuit à partir de l'instruction qui suit dans le programme l'instruction qui a mis en action la procédure. Le même effet produit le passage du contrôle à une instruction || RETURN || (voir le p. 9.7.12). Pour une fonction, cette instruction est le seul moyen de se terminer. Dans ce cas, l'exécution du programme se poursuit à partir du point d'appel de la fonction dans le programme.

Lorsque dans un programme, l'instruction qui précède directement l'en-tête de procédure est autre qu'une instruction || IF|| ou || GO TO|| le contrôle passe à l'instruction qui suit immédiatement la dernière instruction de la procédure, c'est-à-dire que la procédure est sautée.

Il est possible que des procédures soient emboîtées, mais elles ne doivent pas se couper.

Une procédure qui n'est contenue dans aucune autre procédure est dite extérieure, dans le cas contraire elle est dite intérieure.

Il est à remarquer qu'en plus d'entrées principales dont les noms sont cités dans l'instruction de procédure, une procédure peut avoir d'autres entrées qui sont réalisées au moyen de l'instruction || ENTRY|| (voir le p. 9.7.10).

En ce qui concerne les domaines d'action de noms et l'allocation de mémoire pour les variables, les procédures sont analogues aux blocs (voir le p. 9.7.1). Elles diffèrent de ces derniers par le principe d'échange de l'information basé sur la correspondance des arguments (des paramètres effectifs) et des paramètres formels, ainsi que par la méthode d'initialisation. La correspondance entre les arguments (les paramètres effectifs) et les paramètres formels est celle de leurs positions réciproques dans les listes de paramètres. Il faut alors

estimer: 1) la correspondance en nombre des arguments et des paramètres de procédure et 2) la correspondance des types de grandeurs définies par les arguments et les paramètres. La dernière condition veut dire que si un paramètre d'une entrée en procédure est un scalaire, alors l'argument doit être une expression scalaire. Les descripteurs des données de cet argument doivent être conformes aux descripteurs correspondants des paramètres. Si un paramètre représente un tableau (une structure), alors l'argument doit être une expression du type tableaux (les structures). Il est admissible qu'il soit une expression scalaire, à condition de le décrire, pour l'entrée correspondante en procédure, par un descripteur || ENTRY || contenant un descripteur de sa dimension (une description de la structure du paramètre). Les descripteurs des données d'un argument doivent être conformes aux descripteurs des paramètres. Si un paramètre représente une structure, l'argument et le paramètre doivent avoir la même organisation bien que la coïncidence des numéros de niveaux ne soit pas exigée.

Si un paramètre représente une variable du type cellule, l'argument correspondant doit être lui aussi du type cellule, avec la même construction relative, bien que la coïncidence de niveaux ne soit pas exigée.

Si un paramètre est une variable scalaire du type étiquette, l'argument doit être une variable scalaire ou une constante du type étiquette. Si le paramètre représente un tableau du type étiquette, tel doit être l'argument.

Si un paramètre est un paramètre-entrée, alors l'argument doit être un nom d'entrée ou un paramètre-entrée. Dans les deux cas, tous les descripteurs doivent être donnés explicitement et sont obligés à correspondre l'un à l'autre.

Si un paramètre est un paramètre-fichier, l'argument doit être un nom de fichier ou un paramètre-fichier. On n'exige pas que les descripteurs du paramètre-fichier coïncident avec ceux de l'argument.

Un paramètre pour lequel le mode d'allocation de mémoire n'est pas spécifié admet n'importe quel mode d'allocation pour l'argument. S'il existe plusieurs générations de l'argument, le paramètre s'identifie à la génération en usage au moment d'appel.

Pourtant, un paramètre non basé *contrôlé* doit être utilisé toujours avec un argument contrôlé (dans ce cas l'argument doit être un nom des données non indicé à l'allocation contrôlée, voir les pp. 9.6.3, 9.7.14, 9.7.15). Un tel paramètre s'identifie à tout le magasin des allocations pour la variable contrôlée donnée.

Si l'argument est une chaîne ou un tableau, alors dans la procédure appelée la longueur de la chaîne ou les bornes du tableau doivent être déclarées ou bien avec l'utilisation du symbole || * ||, ou bien avec l'indication des bornes ou de la longueur directement

ou sous forme d'une expression qui fournit après le calcul la valeur correspondante (voir le p. 9.6.1, descripteur de dimensions).

Le nombre de dimensions et les bornes d'un tableau-argument ou la longueur d'une chaîne-argument doivent être les mêmes que pour le paramètre correspondant.

Si une procédure est récursive, alors chaque fois que l'appel récursif de la procédure a lieu, pour chaque variable rangée en mémoire selon le mode d'allocation automatique une nouvelle génération est créée.

Du point de vue structural un algorithme donné en PL/1 (un programme PL/1) représente une procédure extérieure ou une suite de procédures extérieures dont l'une que l'on appelle procédure extérieure principale, reçoit la commande de l'extérieur de ce programme PL/1. Dans le cas d'une procédure extérieure, unique, celle-ci est la procédure principale.

9.7.10. Instruction d'entrée en procédure. L'*instruction d'entrée dans une procédure* indique une entrée supplémentaire et a la forme

$$\| x_1 : x_2 : \ldots : x_p : \text{ENTRY } ab; \|,$$

où x_i ($i = 1, 2, \ldots, p$), a et b ont les mêmes significations que dans une instruction de procédure (voir le p. 9.7.9).

Si l'instruction d'entrée a plusieurs étiquettes, chacune d'elles est considérée comme le nom d'une entrée indépendante (avec une étiquette), avec la même liste de paramètres et les mêmes descripteurs de données que dans l'instruction de procédure.

Une instruction d'entrée doit être intérieure pour la procédure dont elle définit une entrée supplémentaire. Cette instruction ne peut être intérieure par rapport à aucun bloc (aucune procédure) contenu dans cette procédure.

9.7.11. Instruction d'appel d'une procédure. L'*instruction d'appel d'une procédure* a la forme

$$\| \text{CALL } xabcd; \|,$$

où x est un identificateur qui coïncide avec l'une des étiquettes de l'instruction de procédure ou d'une instruction d'entrée en procédure, que l'on appelle nom d'entrée en procédure; a est ou bien vide, ou bien une liste ($s_1, s_2, \ldots, s_n$) des arguments (des paramètres effectifs); b est ou bien vide, ou bien une construction de la forme $\|$ TASK $e\|$, dans laquelle le mot réservé $\|$ TASK $\|$ est l'*option de branche* et e est soit vide, soit une liste ($t_1, t_2, \ldots, t_n$) des noms t_i de branches; c est ou bien vide, ou bien une construction de la forme $\|$ EVENT $v\|$ dans laquelle le mot réservé $\|$ EVENT $\|$ est l'*option d'événement*, et v est soit vide, soit une liste ($q_1, q_2, \ldots, q_f$) des noms q_i d'événements; d est soit vide, soit une construction

de la forme || PRIORITY (w)|| dans laquelle le mot réservé || PRIORITY|| est l'*option de priorité* et w est une expression.

Il faut remarquer que les constructions b, c et d dans une instruction d'appel d'une procédure peuvent être mentionnées dans n'importe quel ordre l'une par rapport à l'autre. La présence d'une combinaison quelconque de ces constructions dans une instruction d'appel d'une procédure signifie que, premièrement, au cours de l'exécution de cette instruction il sera créé une branche de nom indiqué dans l'option de branche si la construction b figure dans l'instruction d'appel, de plus, l'utilisation de ce nom permet d'établir la priorité de la branche au moyen de la fonction incorporée respective et de la pseudo-variable || PRIORITY|| (voir le § 9.9), et deuxièmement, les procédures appelante et appelée seront exécutées de manière asynchrone (voir le § 9.3). L'absence de la construction b signifie que la branche créée sera sans nom.

La présence de la construction c dans l'instruction d'appel d'une procédure signifie que l'on introduit une variable logique de nom indiqué dans la liste des noms d'événements de la construction c. Cette variable prend la valeur || '0'B|| lors de l'exécution de l'instruction d'appel de procédure et || '1'B|| après l'exécution de la branche créée par cette instruction d'appel de procédure.

Si l'option || PRIORITY (w)|| est spécifiée dans une instruction d'appel de procédure, alors au cours de l'exécution de l'instruction, la valeur de l'expression w est calculée et mise sous la forme || FIXED m, 0||, où m dépend de la réalisation; la priorité de la branche créée en exécutant l'instruction donnée par rapport à la branche dans laquelle cette instruction est exécutée devient égale à cette valeur de l'expression w. Lorsque la construction d est vide, le niveau prioritaire doit être attribué à la branche au moyern de la pseudo-variable || PRIORITY|| avant l'exécution de l'instruction d'appel de la procédure.

Un argument d'une instruction d'appel de procédure peut être une expression de type quelconque, une constante du type étiquette, un tableau ou un paramètre du type étiquette, un nom d'entrée, un paramètre-entrée, un nom de fichier, un paramètre-fichier, un nom det branche, un paramètre-branche, un nom d'événement, un paramè re-événement, un nom de domaine, un paramètre-domaine, un nom de pointeur, une expression sur les pointeurs (une telle expression doit être soit une variable du type pointeur, soit un appel à une fonction à valeurs du type pointeur), un paramètre-pointeur.

Une instruction d'appel d'une procédure assure l'appel de la procédure de nom x, le remplacement des paramètres formels de la procédure par les arguments (les paramètres effectifs) et le passage du contrôle à cette procédure. Après l'exécution de la procédure le contrôle est remis à l'instruction qui suit l'instruction d'appel

de la procédure (pour plus de détails sur l'exécution d'une procédure, voir le p. 9.7.9).

9.7.12. Instruction de retour. L'*instruction de retour* fait cesser l'exécution de la procédure qui la contient et rend le contrôle à la procédure qui a appelé la procédure donnée. L'instruction de retour a la forme

|| RETURN *a*;||,

où *a* est soit vide, soit une construction de la forme (*w*) dans laquelle *w* est une expression.

Une instruction de retour avec *a* vide s'utilise dans le cas où il est impossible d'appeler la procédure contenant cette instruction au moyen d'un indicateur de fonction (voir le § 9.4). Lorsque l'instruction d'appel d'une procédure contenant une instruction || RETURN|| contient également une option d'événement, i.e. une construction || EVENT *v*||, où *v* est ou bien vide, ou bien une liste (q_1, q_2, . . ., q_n) des noms q_i d'événements, l'exécution de l'instruction de retour aura pour effet l'affectation de la valeur || '1'B|| aux noms de ces événements.

Une instruction de retour contenant une expression *w* ne s'utilise que dans le cas où l'appel de la procédure s'effectue à l'aide d'un indicateur de fonction. L'instruction de retour rend alors le contrôle à l'endroit du programme d'où est venu l'appel de la procédure, et la valeur que fournit la fonction est celle de l'expression *w*.

Si la déclaration de l'entrée par laquelle la procédure a été appelée contient des descripteurs de données, alors avant le retour des valeurs celles-ci subissent des transformations définies par les descripteurs de données.

9.7.13. Instruction de fin. L'*instruction de fin* s'utilise en tant qu'instruction finale d'une boucle, d'un bloc ou d'une procédure et a la forme

|| END *z*;||,

où *z* est soit vide, soit une étiquette coïncidant avec l'un des noms d'entrées dans une boucle, un bloc ou dans une procédure qui contiennent l'instruction de fin donnée.

Si *z* est vide, l'instruction || END|| ferme la boucle, le bloc ou la procédure qui commencent respectivement par l'instruction précédente la plus proche || DO||, || BEGIN|| ou || PROCEDURE||. L'instruction de fin peut fermer (se rapporter à) plusieurs boucles (blocs ou procédures) à la fois.

Les actions d'une instruction de fin terminant une procédure coïncident avec celles d'une instruction || RETURN||. L'action d'une instruction de fin fermant un bloc qui représente un programme

de réaction à une interruption consiste en la transmission du contrôle à l'instruction suivant l'instruction où la situation d'interruption s'est produite. Pour l'action d'une instruction de fin fermant une boucle voir le p. 9.7.5.

9.7.14. Instruction d'allocation de mémoire. L'*instruction d'allocation de mémoire* effectue des réservations de mémoire à différentes grandeurs et peut être de deux types. L'instruction du premier type est de la forme

$$\| \text{ALLOCATE } n_1x_1s_1p_1,\ n_2x_2s_2p_2,\ \ldots,\ n_kx_ks_kp_k\ ;\|,$$

où x_i est un identificateur (un nom de variable contrôlée, d'un tableau ou d'une structure) ; n_i est soit vide, soit un numéro de niveau si x_i représente un nom de structure de niveau supérieur (un nom d'élément de structure différant d'un nom de structure de niveau supérieur peut être indiqué en tant que x_i si l'organisation de toute la sous-structure de nom x_i est identique à celle de la structure définie par l'instruction || DECLARE ||) ; s_i est soit vide, soit un descripteur de dimensions (voir le p. 9.6.1) si x_i est un nom de tableau ; p_i est l'un des descripteurs || BIT || CHARACTER || INITIAL || ou une combinaison de ces descripteurs qui spécifie la grandeur de nom x_i (les descripteurs || BIT || CHARACTER || ne peuvent s'utiliser qu'avec un identificateur déclaré comme binaire ou chaîne de caractères respectivement). L'instruction d'allocation de mémoire du deuxième type a la structure

$$\| \text{ALLOCATE } x_1 \text{ SET } (p_1)\, s_1,\ x_2 \text{ SET } (p_2)\, s_2,\ \ldots,\ x_k \text{ SET } (p_k) S_k\ ;\|,$$

où x_i $(i = 1, 2, \ldots, k)$ est un identificateur (un nom de variable basée), p_i est un identificateur (un nom de variable du type pointeur), s_i est soit vide, soit une construction de la forme || INTO (q_i) || dans laquelle q_i est un identificateur (un nom de variable du type domaine).

Une variable basée x_i peut être un scalaire, un tableau ou une structure de niveau supérieur. L'ordre de succession des constructions || SET (p_i) || et || INTO (q_i) || est arbitraire.

Une instruction d'allocation du premier type fait « descendre » pour tout identificateur x_i une partie de mémoire dans le magasin, et par cela crée une nouvelle génération (un nouvel exemplaire) de la grandeur de nom x_i. La mémoire réservée à la grandeur en question (à la nouvelle génération de cette grandeur) ne peut être libérée qu'à la suite d'exécution d'une instruction || FREE || (voir le p. 9.7.15), et alors toute la pile des valeurs refoulées vers le bas du magasin « monte ». L'étendue de la zone à réserver à une grandeur est définie soit par les bornes de tableau, soit par la longueur de la

chaîne, suivant que l'identificateur en question est un nom de tableau ou de chaîne. Si les bornes du tableau ou la longueur de la chaîne ne sont pas données dans l'instruction d'allocation, elles sont choisies dans l'instruction || DECLARE || qui appartient à la même procédure ou au même bloc que l'instruction d'allocation et qui contient la description de la variable considérée. Si les bornes du tableau ou la longueur de la chaîne sont données par le symbole || * ||, alors la génération actuelle de la variable s'identifie à la génération précédente de cette variable.

Si une instruction d'allocation (ou une instruction || DECLARE ||) contient le descripteur || INITIAL || pour la variable x_i, alors, en exécutant l'instruction d'allocation, on attribue à cette variable la valeur initiale. Si ce descripteur est présent aussi bien dans l'instruction d'allocation que dans l'instruction || DECLARE ||, on utilise le descripteur de l'instruction d'allocation. L'affectation des valeurs initiales à une grandeur peut se faire par l'intermédiaire d'une autre grandeur qui, au moment donné, peut ne pas être située dans la mémoire. Pour savoir si telle ou telle grandeur est déjà rangée en mémoire, on peut se servir de la fonction incorporée || ALLOCATION || (voir le § 9.9).

En exécutant une instruction d'allocation du deuxième type on attribue à la variable du type pointeur p_i ($i = 1, 2, \ldots, k$) la valeur qui situe dans la mémoire une nouvelle génération de la variable basée x_i (la variable p_i n'est pas obligée de coïncider avec la variable pointeur liée à la variable x_i et décrite dans l'instruction || DECLARE ||). Si s_i n'est pas vide, alors la nouvelle génération de la variable basée x_i est stockée dans la partie de mémoire spécifiée par la variable q_i du type domaine. Si cette partie de mémoire ne suffit pas pour y placer la nouvelle génération de la variable x_i, il y aura une interruption par non-respect de la condition domaine (voir le p. 9.8.1).

Si l'identificateur q_i est un nom de tableau, il doit être muni, dans l'instruction d'allocation, d'indices.

Si s_i est vide, une nouvelle génération de la variable x_i est située dans la partie standard de la mémoire prévue par le système d'exploitation.

Le volume de mémoire réservé à une variable basée x_i dépend de ses propriétés spécifiées dans l'instruction || DECLARE || où cette variable est déclarée (aucun descripteur de la variable x_i ne peut être donné dans l'instruction d'allocation). Or, il est interdit de se servir du symbole || * || pour définir les bornes de tableau ou la longueur de chaîne de la variable basée x_i.

Si une variable x_i est déclarée dans l'instruction || DECLARE || avec le descripteur || INITIAL ||, l'affectation des valeurs initiales se fait après le placement de cette variable et après l'affectation d'une valeur à la variable p_i du type pointeur.

9.7.15. Instruction de libération de mémoire. L'*instruction de libération de mémoire* s'utilise pour vider la mémoire occupée par les variables indiquées dans l'instruction. (Cette mémoire s'est trouvée occupée à la suite d'exécution d'une instruction d'allocation contenant les mêmes variables contrôlées.) L'instruction || FREE || et l'instruction correspondante d'allocation doivent appartenir à la même procédure.

L'instruction || FREE || a la forme

$$\| \text{FREE } a_1x_1, a_2x_2, \ldots, a_kx_k ; \|,$$

où x_i ($i = 1, 2, \ldots, k$) est une variable scalaire (un tableau, une structure) contrôlée (basée ou non) qui était placée dans la mémoire par l'instruction correspondante d'allocation, sans participation d'une variable du type domaine; a_i est soit vide si x_i est une variable non basée (un tableau, une structure), soit une construction $\| p_i - >0 \|$ dans laquelle p_i est une variable du type pointeur si x_i est une variable basée (un tableau, une structure).

9.7.16. Instruction d'attente. L'*instruction d'attente* a la forme

$$\| \text{WAIT } x_1, x_2, \ldots, x_n, w ; \|,$$

où x_i ($i = 1, 2, \ldots, n$) est un nom d'événement; w est ou bien une expression scalaire qui doit être transformée en un nombre entier vers le moment d'exécution de l'instruction, ou bien vide.

L'instruction || WAIT || arrête l'exécution de la branche (voir le § 9.3) à laquelle elle appartient jusqu'à ce que les k ($k \leqslant n$) premiers événements dont les noms sont cités dans l'instruction ne soient réalisés *). La quantité k est déterminée par l'expression w. Si $k \leqslant 0$, l'effet de l'instruction || WAIT || est équivalent à celui d'une instruction vide.

Le cas de w vide est équivalent à l'éventualité $k = n$.

9.7.17. Instruction de description de données. L'*instruction de description de données* sert à décrire les propriétés de variables et dans le cas le plus simple a la forme

$$\| \text{DECLARE } n_1x_1y_1^{(1)}y_1^{(2)} \ldots y_1^{(k_1)}, n_2x_2y_2^{(1)}y_2^{(2)} \ldots y_2^{(k_2)}, \ldots$$
$$\ldots, n_mx_my_m^{(1)}y_m^{(2)} \ldots y_m^{(k_m)} ; \|,$$

où x_i ($i = 1, 2, \ldots, m$) est un nom de variable; n_i est soit vide, soit un entier décimal indiquant le niveau du nom x_i (si n_i est vide,

* C'est-à-dire jusqu'à ce que la condition || EVENT (x_i)='1'B || ne soit satisfaite (voir le § 9.9).

le niveau est égal à 1 par omission); $y_i^{(j)}$ ($j = 1, 2, \ldots, k_i$) sont des descripteurs de la variable x_i.

Exemple 9.53. L'instruction || DECLARE X FIXED AUTOMATIC EXTERNAL, Y FLOAT AUTOMATIC EXTERNAL, Z CONTROLLED EXTERNAL;|| déclare que la variable X de niveau 1 est extérieure à allocation automatique, à valeurs réelles décimales à point fixe; la variable Y de niveau 1 est extérieure, à allocation automatique, à valeurs décimales réelles à point flottant; la variable Z de niveau 1 est extérieure, à allocation contrôlée.

On peut réunir les descripteurs communs à plusieurs noms déclarés en mettant ces noms entre parenthèses. Les descripteurs sont notés en dehors des parenthèses.

Exemple 9.54. L'instruction ||DECLARE|| de l'exemple 9.53 est équivalente à l'instruction || DECLARE ((X FIXED, Y FLOAT) AUTOMATIC, Z CONTROLLED) EXTERNAL;||.

9.7.18. Instruction d'entrée-sortie. Les *instructions d'entrée-sortie* réalisent l'échange d'informations entre les mémoires extérieures (bandes ou disques magnétiques, imprimantes, etc.) et la mémoire principale ou calculateur.

Ce sont les instructions || READ || WRITE || REWRITE || GET || PUT || LOCATE ||.

Les données transférées entre mémoires sont mises sous forme de suite d'enregistrements, ces suites s'appelant *fichiers*. A chaque fichier est associé un identificateur appelé *nom de fichier*. Cette correspondance s'établit soit explicitement, à l'aide de l'instruction || OPEN || (voir le p. 9.7.18.1), soit implicitement, en exécutant l'une des instructions suivantes:

|| GET || pour un fichier avec les descripteurs || STREAM || INPUT ||;
|| PUT || pour un fichier avec les descripteurs || STREAM || OUTPUT ||;
|| READ || pour un fichier avec les descripteurs || RECORD || INPUT ||;
|| WRITE || pour un fichier avec les descripteurs || RECORD || OUTPUT ||;
|| REWRITE || pour un fichier avec les descripteurs || RECORD || UPDATE ||;
|| LOCATE || pour un fichier avec les descripteurs || RECORD || OUTPUT || SEQUENTIAL || BUFFERED ||;
|| DELETE || pour un fichier avec les descripteurs || RECORD || DIRECT || UPDATE ||;
|| UNLOCK || pour un fichier avec les descripteurs || RECORD || DIRECT || UPDATE || EXCLUSIVE ||.

Les instructions d'entrée-sortie réalisent l'entrée des enregistrements d'un fichier dans la mémoire avec ou sans transformation de données; les instructions de sortie réalisent la sortie de données de la mémoire vers un fichier en les présentant comme enregistrements.

Le langage PL/1 dispose de deux modes d'entrée-sortie: par enregistrements et par une séquence ininterrompue d'enregistrements. Dans le deuxième cas le fichier est considéré comme une suite unique de symboles appelée *flux*.

Il est clair, donc, qu'en déclarant un fichier avec une instruction || DECLARE ||, on doit le munir par l'un des descripteurs || RECORD || ou || STREAM || (flux). En l'absence de ces descripteurs, on choisit || STREAM || par omission.

On distingue les fichiers à *accès séquentiel* et à *accès direct*. On mentionne le mode d'accès d'un fichier dans une instruction || DECLARE || en mettant l'un des descripteurs || SEQUENTIAL || ou || DIRECT ||. En l'absence de ces descripteurs, c'est le || SEQUENTIAL || qui est choisi par omission.

Le descripteur || SEQUENTIAL || signifie que la lecture, l'écriture ou la modification d'enregistrements d'un fichier muni de ce descripteur se fait dans l'ordre séquentiel, à partir du premier enregistrement jusqu'au dernier. Le descripteur || DIRECT || signifie que la lecture, l'écriture ou la modification d'enregistrements d'un fichier dont la déclaration contient ce descripteur se fait de façon sélective, l'enregistrement à traiter étant indiqué. Autrement dit, un fichier à accès direct permet l'entrée-sortie de n'importe quel enregistrement selon le mot-clé (voir le p. 9.6.8).

9.7.18.1. Instruction d'ouverture de fichiers. L'*instruction d'ouverture de fichiers* a la forme

$$|| \text{OPEN } a_1, a_2, \ldots, a_n ; ||,$$

où a_i $(i = 1, 2, \ldots, n)$ est un ensemble de descripteurs. A cet ensemble peuvent appartenir: un descripteur || FILE (x) ||, où x est un nom de fichier; l'un des descripteurs || INPUT || OUTPUT || UPDATE ||; l'un des descripteurs || STREAM || RECORD ||; l'un des descripteurs || DIRECT || SEQUENTIAL ||; l'un des descripteurs || BUFFERED || EXCLUSIVE || UNBUFFERED ||; le descripteur || KEYED (m) ||, où m est un entier décimal; le descripteur || BACKWARDS ||; le descripteur || PRINT ||; le descripteur || LINESIZE (y) ||, où y est une expression à convertir en un nombre entier décimal positif; le descripteur || PAGESIZE (y) ||, où y est une expression à convertir en un nombre entier décimal positif.

Le descripteur || FILE (x) || indique le fichier à ouvrir. Ce descripteur doit figurer une seule fois dans chaque groupe de descripteurs a_i.

Le descripteur || LINESIZE (y)|| ne peut être donné que pour un fichier du type || STREAM PRINT||. La quantité y indique la longueur d'une ligne du fichier donné.

Le descripteur || PAGESIZE (y)|| ne peut être donné que pour un fichier du type || STREAM PRINT||. La quantité y indique le nombre de lignes d'une page du fichier donné.

Tous les autres descripteurs d'un groupe a_i complètent les descripteurs donnés dans la déclaration d'un fichier.

9.7.18.2. Instruction de fermeture de fichiers. L *instruction de fermeture de fichiers* est destinée à supprimer la correspondance entre les noms de fichiers et les fichiers mêmes indiqués dans l'instruction, i.e. à fermer ces fichiers. L'instruction || CLOSE|| a la forme

$$|| \text{CLOSE } a_1, a_2, \ldots, a_n ; ||,$$

où a_i ($i = 1, 2, \ldots, n$) est le descripteur || FILE (x_i)|| dans lequel x_i est un nom de fichier.

On ne peut employer une instruction || CLOSE|| que pour fermer les fichiers qu'on a ouverts dans la même branche. Lorsqu'une branche ne contient pas d'instruction || CLOSE|| pour fermer un fichier qui y a été ouvert, ce fichier sera fermé automatiquement après l'exécution de la branche.

Si, au cours de l'exécution d'une branche contenant une instruction || CLOSE||, on a bloqué certains enregistrements d'un fichier cité dans l'un des descripteurs de cette instruction, alors après l'exécution de l'instruction || CLOSE|| ces enregistrements seront débloqués.

On peut rouvrir un fichier fermé.

9.7.18.3. Liste d'entrée-sortie. Aux instructions d'entrée-sortie est liée la notion de *liste d'entrée-sortie* qui détermine les zones de mémoire utilisées pour l'entrée-sortie.

Une liste d'entrée-sortie a la forme

$$(a_1, a_2, \ldots, a_n),$$

où a_i ($i = 1, 2, \ldots, n$) est un élément de liste qui peut être représenté par une variable (simple ou indicée), un nom de tableau ou de structure de niveau supérieur, un élément de boucle. Les variables et les éléments de tableaux ou de structures de niveau supérieur doivent être du type arithmétique ou être des chaînes. On entend par *élément de boucle* une construction de la forme

$$(b_1, b_2, \ldots, b_{k_i}, y_i),$$

où b_j ($j = 1, 2, \ldots, k$) est soit une variable, soit un nom de tableau ou de structure de niveau supérieur, soit un élément de boucle;

y_i est un en-tête de boucle, i.e. une construction qui a dans le cas général la forme

$$\| \text{DO } z_i = t_i \text{ TO } s_i \text{ BY } q_i \text{ WHILE } t \|$$

(voir le p. 9.7.5).

Si un élément de liste d'entrée-sortie représente une variable, cela signifie qu'il faut entrer (sortir) la valeur de cette variable.

Si un élément de liste d'entrée-sortie représente un nom de tableau (de structure de niveau supérieur), il s'agit de l'entrée-sortie de tous les éléments du tableau (de la structure) dans l'ordre de leur disposition dans le tableau (ou conformément à l'organisation de la structure et à l'ordre de succession des éléments des tableaux contenus dans la structure).

Si un élément de liste d'entrée-sortie représente un élément de boucle, les opérations d'entrée-sortie portent sur les valeurs des grandeurs indiquées dans la liste de l'élément de boucle, l'ordre de leur succession devant correspondre à la variation cyclique de la variable z_i.

EXEMPLE 9.55. L'instruction || GET LIST ((A (I, J) DO I = 1 TO 2) DO J = 3 TO 4); || est équivalente à la construction

```
|| DO J = 3 TO 4;
   DO I = 1 TO 2;
   GET LIST (A (I, J));
   END;
   END; ||
```

qui fournit les éléments du tableau A dans l'ordre suivant: A (1,3), A (2,3), A (1,4), A (2,4).

EXEMPLE 9.56. Soit X la structure

```
|| 1X (5),
      2Y,
      2Z ||.
```

Alors l'instruction || PUT FILE (A) LIST (X); || définira l'ordre suivant des éléments en sortie: A.B (1), A.C (1), A.B (2), A.C (2), A.B (3), A.C (3), A.B (4), A.C (4), A.B (5), A.C (5).

9.7.18.4. Entrée-sortie d'un flux de données. Il y a trois méthodes d'entrée-sortie d'un flux de données: l'entrée-sortie commandée par une liste, l'entrée-sortie commandée par les données et l'entrée-sortie commandée par l'édition.

L'*entrée-sortie commandée par une liste* est définie par le spécificateur de contrôle de la forme

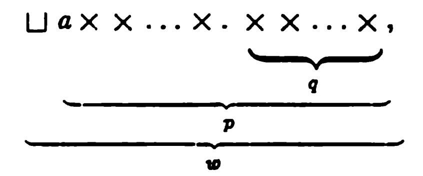
$$\| \text{LIST } c \|,$$

où c est une liste d'entrée-sortie. Elle précise la zone de mémoire où les données seront placées (extraites) (voir le p. 9.7.18.3).

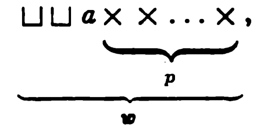
Les données dans un flux commandé par une liste peuvent être des nombres réels et complexes, des chaînes de caractères et des chaînes binaires, le « vide ».

La forme de représentation des nombres et chaînes dans un flux est décrite plus bas.

Un nombre réel décimal à point fixe, avec la précision (p, q), où p est le nombre total de chiffres significatifs dans le champ de données et q le nombre de chiffres après le point décimal, se présente comme



si $0 < q \leqslant p$, ou comme



si $q = 0$. Ici $\times$ désigne un chiffre décimal quelconque, a désigne le symbole $\| - \|$ si le nombre est négatif, et $\| + \|$ dans le cas contraire.

Un nombre réel décimal à point fixe, de précision (p, q) et avec un facteur d'échelle, est représenté dans un flux comme

$$\underbrace{\sqcup \underbrace{a \times \times \ldots \times}_{p} \text{F} - yy \ldots y}_{w},$$

où a est le symbole $\| - \|$ si le nombre est négatif, et $\| + \|$ dans le cas contraire; $\times$ représente un chiffre décimal quelconque; $yy \ldots \ldots y = -q$ est le facteur d'échelle; $w = p + 3 + n$, où n est le nombre de chiffres nécessaires à la représentation de q.

Un nombre réel binaire à point fixe doit être converti en un nombre décimal à point fixe et écrit dans le flux sous cette forme.

Un nombre réel décimal à point flottant est représenté sous la forme définie par le format E (w, d, s), où d est le nombre de posi-

tions dans le champ de données à droite du point décimal $w = p + 6$, $d = p - 1$ et $s = p$ (voir le p. 9.7.18.5).

Un nombre réel binaire à point flottant doit être converti en un nombre décimal à point flottant et écrit sous cette forme.

Un nombre complexe est représenté comme deux nombres consécutifs réels, le premier (à gauche) formant la partie réelle, le deuxième, la partie imaginaire. La partie imaginaire est précédée d'un signe (on met « plus » si la valeur de la partie imaginaire est positive ou nulle, et « moins » dans le cas contraire) et suivie du symbole || I ||.

Une chaîne s'écrit dans un flux sans aucune transformation si le fichier possède le descripteur || PRINT || (imprimer). Dans ce cas la taille du champ coïncide avec celle de la chaîne. Si le fichier n'a pas de descripteur || PRINT ||, alors la chaîne écrite dans le flux est mise entre guillemets, chaque guillemet intérieur étant doublé. (La taille du champ sera égale à la longueur de la chaîne plus le nombre de guillemets ajoutés.) Une chaîne binaire s'écrit comme une suite de zéros et d'unités mise entre guillemets et suivie du symbole || B ||. La taille du champ est $w = p + 3$, où p est le nombre de zéros et d'unités dans la chaîne binaire.

Deux données dans un flux sont séparées par un blanc ou par une virgule. Le séparateur peut être, à son tour, précédé et suivi d'un nombre arbitraire de blancs.

Un champ vide dans un flux peut être représenté soit par deux virgules consécutives, soit par un groupe de blancs mis entre deux virgules.

Un flux se termine par le symbole || ; ||.

L'*entrée-sortie commandée par les données* est définie par le spécificateur de contrôle de la forme

$$\| \text{DATA } c \|,$$

où c est une liste d'entrée-sortie. Elle spécifie, comme avant la zone de mémoire, où les données doivent être stockées (voir le p. 9.7.18.3).

Les données dans un flux sont mises soit sous la forme

$$x_1 = y_1 \sqcup x_2 = y_2 \sqcup \ldots \sqcup x_n = y_n,$$

soit sous la forme

$$x_1 = y_1, \quad x_2 = y_2, \ldots, x_n = y_n,$$

où x_i est une variable scalaire indicée, les indices étant des entiers décimaux, y_i est un nombre réel décimal (à point fixe ou flottant), ou un nombre complexe, ou une chaîne de caractères ou enfin une chaîne binaire. La forme de représentation de y_i coïncide avec la forme de représentation des données dans un flux commandé par une liste.

L'*entrée-sortie commandée par l'édition* est définie par le spécificateur de contrôle ayant la forme

$$\| \text{EDIT } c_1 f_1 c_2 f_2 \ldots c_n f_n \|,$$

où c_i ($i = 1, 2, \ldots, n$) est une liste d'entrée-sortie, f_i une liste de formats.

Dans le cas le plus simple, la liste de formats a la forme

$$(n_1 a_1, n_2 a_2, \ldots, n_k a_k),$$

où a_j est un élément de format; n_j un facteur de répétition (un entier décimal ou une expression mise entre parenthèses) qui montre que l'élément correspondant de format a_j doit être utilisé n_j fois de suite si $n_j > 0$. Si $n_j \leqslant 0$ ledit élément de format est sauté. Lorsque le facteur de répétition est donné par une expression, sa valeur se calcule et se transforme en un entier pour chaque appel du format.

On distingue deux types d'éléments de format: les éléments de format de données (voir le p. 9.7.18.5) et les éléments de contrôle (voir le p. 9.7.18.6).

Dans un cas plus compliqué, la liste de formats est de la forme

$$(n_1 b_1, n_2 b_2, \ldots, n_m b_m),$$

où b_j ($j = 1, 2, \ldots, m$) est une liste de formats, n_j un facteur de répétition.

9.7.18.5. Types de format de données. Ces formats décrivent la forme de représentation de données dans un flux de données.

On distingue six types de format destinés respectivement à représenter: un nombre décimal à point fixe (symbole || F||); un nombre décimal à point flottant (symbole || E||); un nombre complexe (symbole || C||); une chaîne de caractères (symbole || A||); une chaîne binaire (symbole || B||) et un format à schéma (symbole || P||).

Le *nombre décimal à point fixe* dans un flux est décrit par l'un des formats suivants: || F (w)|| F (w, d)|| F (w, d, p)||, où w est le champ réservé à la notation du nombre, d, le nombre de positions après le point décimal, p, un facteur d'échelle.

Au format F (w, d) correspond la représentation du nombre dans le flux sous la forme

$$\underbrace{\sqcup\sqcup \ldots \sqcup a \times \times \ldots \times . \underbrace{\times \times \ldots \times 00 \ldots 0}_{d}}_{w}$$

ou sous la forme

$$\underbrace{\sqcup\sqcup\sqcup \ldots \sqcup a \times \times \ldots \underbrace{\times \times \times \ldots \times 000 \ldots 0}_{d}}_{w},$$

où × est un chiffre décimal quelconque, a est soit le signe « moins » si le nombre est négatif, soit le symbole || ⊔ || si ce n'est pas le cas.

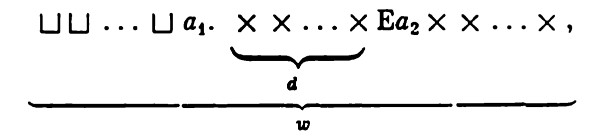
Le format F (w) est considéré comme un format F (w, d) pour $d = 0$.

Le format F (w, d, p) contient un facteur d'échelle qui indique que le nombre donné doit être multiplié en sortie par 10^p.

En sortie, le nombre décimal à point fixe est représenté selon le format et stocké dans la partie droite du champ.

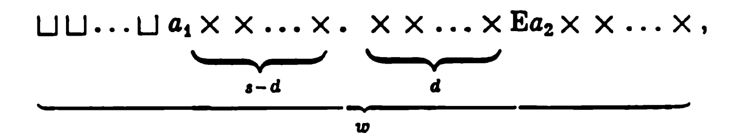
Le *nombre décimal à point flottant* dans le flux est décrit par l'un des formats suivants : || E (w, d)|| E (w, d, s)||, où s est la quantité de chiffres significatifs du nombre donné.

Le format E (w, d) décrit le nombre décimal dans le flux sous la forme



où a_1 est le symbole || —|| si le nombre est négatif, et le symbole || +|| s'il est positif ou nul, a_2 est l'un des symboles || —|| +||; × est un chiffre décimal quelconque (le chiffre décimal qui suit le point est non nul), a_2 ×× . . . × est l'exposant du nombre (choisi de façon que le chiffre après le point décimal soit non nul).

Au format E (w, d, s) correspond la représentation du nombre décimal dans le flux sous la forme



où a_1, a_2 et × ont la même signification que pour le format E (w, d).

Dans les deux cas on a $w \geqslant s + n + 4$, où n est le nombre de chiffres de l'exposant.

Le *nombre complexe* dans le flux est décrit par le format C (z_1, z_2), où z_1 (z_2) est un format du type F ou E. Si z_1 et z_2 est le même format, alors le format du nombre complexe peut être mis sous la forme C (z_1).

Au format C (z_1, z_2) dans le flux correspond une représentation de deux nombres décimaux voisins séparés par le symbole || ,||. Le premier de ces nombres est représenté selon le format z_1, le deuxième, selon le format z_2.

En sortie, le nombre décimal à point flottant dans le flux est représenté conformément au format E (w, d, s), où $w = s + n + 4$.

On peut décrire les nombres décimaux dans le flux par un format P de la forme

P'Z',

où Z est le format d'un nombre (voir le p. 9.6.1). En entrée, Z décrit la forme du nombre; en sortie, la valeur de l'élément de liste d'entrée-sortie est ramenée à la forme imposée par le format Z. Les champs numériques binaires ont après la sortie une représentation symbolique.

Une *chaîne binaire* dans le flux est décrite par le format B (*w*) et a une représentation symbolique. Si *w* dépasse la longueur de la chaîne, celle-ci est complétée à droite par des symboles || ⊔ || jusqu'à la taille de *w*.

En entrée, une chaîne de caractères se transforme en une chaîne binaire; en sortie, il y a une conversion inverse, selon les règles données en 9.5.3. Une chaîne dépassant la longueur prévue est tronquée à droite. Les positions vides sont remplies par des blancs || ⊔ ||.

Une *chaîne de caractères* dans le flux est décrite par l'un des formats || A (*w*) || P'Z' ||, où Z est un format d'une chaîne de caractères (voir le p. 9.6.1).

9.7.18.6. **Formats avec contrôle.** Ce sont le format X et les formats pour l'impression.

Le format X spécifie les intervalles et a la forme

X (*w*),

où *w* est un entier positif.

En entrée, le format X signifie que les *w* symboles rencontrés du flux de données sont à sauter. En sortie, il signifie qu'il faut intercaler dans le flux *w* blancs.

Les formats pour l'impression ne peuvent être utilisés que pour un fichier muni de descripteurs || STREAM || PRINT ||. Un tel fichier se décompose en pages dont les lignes sont disposées l'une au-dessous de l'autre avec des interlignes. La première ligne de n'importe quelle page porte toujours le numéro un. Il y a quatre formats de ce type : || PAGE || SKIP (*w*) || LINE (*w*) || COLUMN (*w*) ||, où *w* est un entier positif ou une expression. (Si c'est une expression, on la transforme en un entier avant d'appeler le format.)

Le format || PAGE || signifie que le flux doit s'avancer jusqu'à une nouvelle page.

Le format || LINE (*w*) || signifie que la page doit s'avancer d'un certain nombre de lignes de façon que les données soient imprimées sur la ligne numéro *w*.

Le format || SKIP (*w*) || signifie qu'il faut sauter $w - 1$ lignes de la page (si $w = 1$, on peut omettre ce nombre dans le format).

Le format || COLUMN (w)|| signifie qu'il faut intercaler dans le flux $w - 1$ symboles || ⊔ || (w doit être inférieur à la taille d'une ligne).

Le format || R (z)|| est encore un format à contrôle; ici z est soit une étiquette d'une instruction de format (voir le p. 9.7.18.7), soit une variable du type étiquette dont les valeurs sont des étiquettes d'instructions de format.

L'option || R (z)|| signifie que le format courant à utiliser en entrée-sortie doit être pris dans l'instruction de format d'étiquette z.

9.7.18.7. Instruction de formats. L'*instruction de format* est utilisée pour l'entrée-sortie de données d'un flux commandé par l'édition. Cette instruction contient une liste d'éléments de format et a la forme

$$\| x_1 : x_2 : \ldots : x_n \text{ FORMAT } y ; \|,$$

où x_i ($i = 1, 2, \ldots, n$) est une étiquette, y une liste de formats (voir les pp. 9.7.18.4, 9.7.18.5, 9.7.18.6).

L'instruction || FORMAT|| est une instruction non exécutable. Cela veut dire que, si le contrôle vient à une instruction || FORMAT||, celle-ci sera sautée, et le contrôle passera à l'instruction suivante.

L'instruction || FORMAT|| est utilisée par les instructions d'entrée-sortie d'un flux de données commandé par l'édition (voir les pp. 9.7.18.4, 9.7.18.8, 9.7.18.9) et contenant un format de contrôle || R (z)||.

9.7.18.8. Instruction d'entrée d'un flux de données. Cette instruction a la forme

$$\| \text{GET FILE } (x)\ zt ; \|,$$

où x est un nom de fichier; t est ou bien vide ou l'option || COPY||; z est un spécificateur de contrôle.

L'option || COPY|| indique que les données introduites doivent être écrites sans modification sur un fichier standard défini par le mot réservé || SYSPRINT|| et destiné à l'impression. (L'option || COPY|| s'utilise en général pour le contrôle d'entrée.)

Si l'instruction d'entrée ne contient pas l'option || FILE (x)||, alors l'entrée sera réalisée d'un fichier standard défini par le mot réservé || SYSIN||.

Une instruction d'entrée orientée vers un flux de données assure l'introduction des données à partir du fichier de nom x ou du fichier || SYSIN|| qui doit être ouvert vers le moment d'entrée. L'ouverture du fichier se fait par une instruction || OPEN|| (voir le p. 9.7.18.1) ou bien, si le fichier n'a pas été encore ouvert, il s'ouvre lors de

l'exécution de l'instruction d'entrée orientée vers un flux de données.

Le spécificateur de contrôle *z* indique le mode de contrôle d'entrée-sortie.

Si le spécificateur *z* a la forme || LIST *c*||, où *c* est une liste d'entrée-sortie, alors le flux de données est commandé par une liste. (Pour la forme de représentation de données dans un tel flux, voir le p. 9.7.18.4.)

Dans ce cas, en entrée, une donnée est lue dans le flux de données, transformée selon les caractéristiques de l'élément correspondant de la liste d'entrée-sortie et affectée comme valeur à cet élément s'il représente une variable réelle simple.

Si l'élément de liste d'entrée-sortie est une variable complexe, on lui attribue une valeur composée de deux données successives lues dans le flux de données.

Si l'élément de liste d'entrée-sortie représente un tableau, alors la première donnée du flux qui correspond à la grandeur en question est attribuée au premier élément de tableau, la deuxième donnée, au deuxième élément de tableau, etc. en tenant compte du rangement « par lignes ».

Un nom de structure dans la liste d'entrée-sortie est équivalent à la liste des variables scalaires et des tableaux qu'elle contient, dans l'ordre indiqué dans la déclaration de la structure.

Si une variable scalaire de la liste d'entrée-sortie est du type chaîne, et si l'élément correspondant du flux de données est une chaîne de caractères, alors les guillemets entre lesquels se trouvent les symboles de cette dernière sont supprimés et les symboles mêmes sont interprétés comme ceux d'une chaîne qui est attribuée à la variable de la liste d'entrée-sortie.

Si le spécificateur *z* a la forme || DATA *c*||, où *c* est une liste d'entrée-sortie, alors le flux de données est un flux commandé par les données. (Pour la forme de représentation de données dans un tel flux, voir le p. 9.7.18.4.)

Dans ce cas, l'exécution de l'instruction d'entrée orientée vers le flux de données se ramène à la lecture des affectations du flux et à l'identification de leurs parties gauches avec les éléments de la liste d'entrée-sortie, après quoi les parties droites sont affectées aux éléments correspondants de la liste d'entrée-sortie.

Si un élément de la liste d'entrée-sortie représente un tableau, le flux de données peut contenir des références indicées à ce tableau. Il se peut que le flux ne contienne pas tous les éléments du tableau. Dans ce dernier cas, les modifications concerneront uniquement les valeurs des éléments du tableau auxquels correspondent les éléments respectifs du flux.

Lorsque pour certains éléments de la liste d'entrée-sortie il n'existe pas d'éléments correspondants dans le flux, il se produit une situation || NAME|| (voir le p. 9.8.1).

Si le spécificateur z a la forme || EDIT $c_1f_1c_2f_2 \ldots c_nf_n$||, où c_i ($i = 1, 2, \ldots, n$) est une liste d'entrée-sortie, f_i est une liste de format, alors le flux de données est commandé par l'édition.

(Pour la forme de représentation de données dans un tel flux, voir le p. 9.7.18.4.)

Dans ce cas, lors de l'exécution de l'instruction d'entrée d'un flux de données, les données d'un fichier sont lues dans l'ordre séquentiel, transformées selon les caractéristiques des éléments de la liste d'entrée-sortie (voir les pp. 9.5.1, 9.5.3), disposées selon les éléments des formats donnés et affectées aux éléments correspondants de la liste d'entrée-sortie.

9.7.18.9. Instruction de sortie d'un flux de données. Cette instruction a la forme

|| PUT FILE (x) $ztsq$||,

où x est un nom de fichier; z un spécificateur de contrôle; t est soit vide, soit l'option || PAGE ||; s est soit vide, soit l'option || SKIP (w)|| dans laquelle w est un entier positif; q est soit vide, soit l'option || LINE (w)||.

Les options t, s et q dans l'instruction de sortie peuvent être spécifiées dans n'importe quel ordre. On les utilise pour l'impression de données comme une information de commande (voir le p. 9.7.18.6).

L'option || FILE (x)|| peut ne pas être présente dans une instruction de sortie, dans ce cas la sortie est réalisée sur le fichier standard défini par le mot réservé || SYSPRINT||.

Lors de l'exécution de l'instruction de sortie orientée vers un flux il se produit l'ouverture du fichier de nom x (ou du fichier || SYSPRINT||), s'il n'a pas été encore ouvert. L'ouverture d'un fichier se fait en général à la suite d'exécution d'une instruction || OPEN|| (voir le p. 9.7.18.1).

Une instruction de sortie orientée vers un flux de données réalise la sortie des valeurs des éléments de la liste d'entrée-sortie spécifiée par le spécificateur z. Ces valeurs sont complètement définies par les propriétés des éléments de la liste d'entrée-sortie. La forme de représentation des valeurs des éléments de cette liste est déterminée par celle des données du flux, si z est l'un des spécificateurs || LIST c|| DATA c|| (voir le p. 9.7.18.4).

Lorsque le spécificateur z a la forme || EDIT $c_1f_1 \ldots c_nf_n$||, la forme de représentation des valeurs des éléments des listes d'entrée-sortie c_i ($i = 1, 2, \ldots, n$) est déterminée respectivement par les éléments de formats $f_1, f_2, \ldots, f_n$ (voir le p. 9.7.18.4). Dans ce dernier cas, en sortie sur l'imprimante, l'édition se fait non seulement selon les options s et q, mais aussi compte tenu des éléments de contrôle (voir le p. 9.7.18.5) qui figurent sur les listes d'éléments de format du spécificateur z.

9.7.18.10. Entrée-sortie orientée vers l'enregistrement. En ce mode d'entrée-sortie, chaque fichier est considéré comme une suite d'enregistrements. Dans le cas général, un enregistrement représente une construction de la forme

$$ab,$$

où a est soit vide, soit une suite de symboles appelée mot-clé, b est une donnée. Elle peut représenter un nombre réel (à point fixe ou flottant), un nombre complexe (un couple de nombres réels séparés par une virgule), une chaîne de caractères ou binaire.

La taille du champ réservé à l'élément de donnée b est déterminée par les propriétés de la grandeur à laquelle cet élément est attribué comme valeur (en entrée) ou de la grandeur dont la valeur sera représentée comme enregistrement (en sortie).

Si a est non vide, la déclaration d'un fichier doit avoir le descripteur || KEY || (voir le p. 9.6.8) qui indique la longueur du mot-clé en symboles. (Dans ce cas le fichier est dit à accès direct.)

L'entrée-sortie orientée vers l'enregistrement peut être réalisée par les instructions || READ || WRITE || LOCATE || REWRITE || DELETE ||. Chacune d'elles admet le mode d'accès séquentiel ou direct. L'instruction d'entrée-sortie sera respectivement dite à accès séquentiel ou direct.

Avant de passer à une étude détaillée des instructions d'entrée-sortie orientées vers l'enregistrement, faisons quelques remarques d'ordre général.

Ces instructions peuvent contenir les options suivantes: || INTO (x) || et || FROM (x) ||, où x est une variable; || KEY (x) ||, où x est une expression; || KEYTO (x) ||, où x est une variable du type chaîne de caractères; || SET (x) ||, où x est une variable du type pointeur; || IGNORE (x) ||, où x est une expression; || EVENT (x) ||, où x est une variable du type événement; || KEY FROM (x) ||, où x est une expression.

L'expression x qui se rencontre dans les options énumérées doit être transformée, vers le moment d'exécution d'une instruction d'entrée-sortie qui contient l'option respective, en un entier décimal non signé.

L'*option* || INTO (x) || s'utilise dans une instruction || READ || et indique que l'enregistrement en entrée doit être attribué à la variable x non indicée de niveau 1.

L'*option* || KEY (x) || s'utilise obligatoirement dans une instruction d'entrée-sortie à accès direct. L'expression x transformée en entier non signé détermine l'enregistrement à introduire.

L'*option* || EVENT (x) || attribue à la variable x la valeur || '0'B || pour le temps d'exécution de l'instruction d'entrée-sortie et || '1'B || après l'exécution.

L'*option* || SET (x) || s'utilise dans une instruction d'entrée et indique que l'enregistrement doit être stocké dans un tampon et que son identificateur sera attribué à la variable x.

L'*option* || IGNORE (x) || peut être donnée dans une instruction d'entrée à accès séquentiel appliquée à un fichier pour lequel sont déclarés les descripteurs || INPUT || ou || UPDATE ||. Elle signifie qu'il faut sauter x enregistrements. Autrement dit, l'instruction d'entrée suivante, si elle appelle le même fichier, aura l'accès au ($x + 1$)-ème enregistrement.

L'*option* || FROM (x) || s'utilise en sortie et indique que la valeur de la variable x non indicée de niveau 1 est à enregistrer sur le fichier.

L'*option* || KEY FROM (x) || indique que la valeur de l'expression x qui représente un entier décimal non signé doit être transformée en une chaîne de caractères et représentée sous la forme du mot-clé de l'enregistrement à placer dans un fichier.

9.7.18.11. Instructions d'entrée-sortie à accès séquentiel. L'*instruction d'entrée dans le tampon* à partir d'un fichier déclaré avec le nom x et les descripteurs || BUFFERED || INPUT || SEQUENTIAL || (voir le p. 9.6.8) peut avoir l'une des formes suivantes :

a) || READ FILE (x) INTO (y) z ; || ;

b) || READ FILE (x) SET (y) z ; ||,

où z est soit vide, soit l'option || KEYTO (t) || ;

c) || READ FILE (x) y ; ||,

où y est soit vide, soit l'option || IGNORE (t) || ;

d) || READ FILE (x) INTO (y) KEY (z) ; || ;

e) || READ FILE (x) SET (y) KEY (z) ; ||.

Les actions accomplies par chacune de ces instructions sont complètement déterminées par les options qui en font partie (voir le p. 9.7.18.10) et par les propriétés des grandeurs figurant dans ces options.

EXEMPLE 9.57. Supposons qu'une instruction || DECLARE || déclare que : 1) ALPHA est un nom de fichier à accès séquentiel pour l'entrée avec tampon ; 2) X et Y sont des variables du type chaîne, A est une variable contrôlée.

Dans ces hypothèses, à la suite d'exécution de l'instruction || READ FILE (ALPHA) SET (A) KEYTO (X) ; ||, les actions suivantes seront accomplies : premièrement, si le fichier de nom ALPHA n'a pas été encore ouvert vers le moment d'exécution de cette instruction, il le sera implicitement ; deuxièmement, l'enregistrement sera transféré, la variable pointeur A prendra pour

valeur l'identificateur de cet enregistrement de sorte que l'on puisse l'identifier dans le tampon (cela signifie en fait que l'enregistrement est stocké dans la mémoire en tant que valeur de la variable Y; ainsi, chaque appel de la variable Y sera un appel de l'enregistrement dans le tampon); troisièmement, le mot-clé de l'enregistrement sera attribué comme valeur à la variable X.

L'instruction d'entrée dans le tampon avec renouvellement. L'instruction d'entrée à partir d'un fichier déclaré avec le nom x et les descripteurs || BUFFERED || SEQUENTIAL || UPDATE || (voir le p. 9.6.8) peut avoir l'une des formes suivantes:

a) || READ FILE (x) INTO (y) z; ||;

b) || READ FILE (x) SET (y) z; ||,

où z est soit vide, soit l'option || KEYTO (t) ||;

c) || READ FILE (x) z; ||,

où z est soit vide, soit l'option || IGNORE (y) ||;

d) || READ FILE (x) INTO (y) KEY (z); ||;

e) || READ FILE (x) SET (y) KEY (z); ||.

Les actions accomplies à la suite d'exécution de chacune des instructions énumérées sont déterminées par les options qui en font partie (voir le p. 9.7.18.10) et par les propriétés des variables figurant dans ces options.

L'instruction d'entrée sans tampon. L'instruction d'entrée à partir d'un fichier déclaré avec le nom x et les descripteurs || UNBUFFERED |||| SEQUENTIAL || INPUT || (voir le p. 9.6.8) peut avoir l'une des formes suivantes:

a) || READ FILE (x) INTO (y) zt; ||,

où z est soit vide, soit l'option || KEYTO (s) ||; t est soit vide, soit l'option || EVENT (q) ||;

b) || READ FILE (x) yz; ||,

où y est soit vide, soit l'option || IGNORE (t) ||, : z est soit vide, soit l'option || EVENT (s) ||;

c) || READ FILE (x) INTO (y) KEY (z) t; ||,

où t est soit vide, soit l'option || EVENT (q) ||.

Les actions accomplies à la suite d'exécution des instructions énumérées sont déterminées par les options qui en font partie et par les propriétés des grandeurs figurant dans ces options (voir le p. 9.7.18.10).

L'instruction d'entrée sans tampon avec renouvellement. L'instruction d'entrée à partir d'un fichier déclaré avec le nom x et les descripteurs ‖ SEQUENTIAL ‖ UNBUFFERED ‖ UPDATE ‖ (voir le p. 9.6.8) peut avoir l'une des formes suivantes:

a) ‖ READ FILE (x) FROM (y) zt; ‖,

où z est soit vide, soit l'option ‖ KEYTO (s) ‖; t est soit vide, soit l'option ‖ EVENT (q) ‖;

b) ‖ READ FILE (x) yz; ‖,

où y est soit vide, soit l'option ‖ IGNORE (t) ‖, z est soit vide, soit l'option ‖ EVENT (q) ‖;

c) ‖ READ FILE (x) INTO (y) KEY (z) t; ‖,

où t est soit vide, soit l'option ‖ EVENT (q) ‖.

Les actions accomplies à la suite d'exécution des instructions énumérées sont déterminées par les options qui en font partie et par les propriétés des grandeurs figurant dans ces options (voir le p. 9.7.18.10).

L'instruction de sortie du tampon. L'instruction de sortie sur un fichier déclaré avec le nom x et les descripteurs ‖ SEQUENTIAL ‖ BUFFERED ‖ OUTPUT ‖ (voir le p. 9.6.8) peut avoir l'une des formes suivantes:

‖ WRITE FILE (x) FROM (y) z; ‖,

où z est soit vide, soit l'option ‖ KEY FROM (t) ‖;

‖ LOCATE y FILE (x) SET (z) t ‖,

où y est un nom de variable, de tableau ou de structure de niveau supérieur; t est soit vide, soit l'option ‖ KEY FROM (s) ‖.

De même que pour les instructions d'entrée, les actions accomplies à la suite d'exécution des instructions de sortie sont déterminées par les options utilisées et par les propriétés des grandeurs figurant dans ces options. La même remarque concerne toutes les autres instructions de sortie à accès séquentiel.

L'instruction de sortie sans tampon. L'instruction de sortie sur un fichier déclaré avec le nom x et les descripteurs ‖ SEQUENTIAL ‖ UNBUFFERED ‖ OUTPUT ‖ (voir le p. 9.6.8) peut avoir la forme

‖ WRITE FILE (x) FROM (y) zt; ‖,

où z est soit vide, soit l'option ‖ KEY FROM (s) ‖; t soit vide, soit l'option ‖ EVENT (q) ‖.

L'instruction de sortie du tampon avec renouvellement. L'instruction de sortie sur un fichier déclaré avec le nom x et les descripteurs

|| SEQUENTIAL || BUFFERED || UPDATE || (voir le p. 9.6.8) peut avoir la forme

|| REWRITE FILE (x); ||

ou la forme

|| REWRITE FILE (x) FROM (y); ||.

L'instruction de sortie sans tampon avec renouvellement. L'instruction de sortie sur un fichier déclaré avec le nom x et les descripteurs || SEQUENTIAL || UNBUFFERED || UPDATE || a la forme

|| REWRITE FILE (x) FROM (y) z; ||,

où z est soit vide, soit l'option || EVENT (t) ||.

9.7.18.12. Instruction d'entrée-sortie à accès direct. L'instruction d'entrée déclarée avec le nom x et les descripteurs || INPUT || (ou || UPDATE ||) et || DIRECT || a la forme

|| READ FILE (x) INTO (y) KEY (z) t; ||,

où t est soit vide, soit l'option || EVENT (s) ||.

L'instruction de sortie sur un fichier déclaré avec le nom x et les descripteurs || OUTPUT || (ou || UPDATE ||) et || DIRECT || a la forme

|| WRITE FILE (x) FROM (y) KEY FROM (z) t; ||,

où t est soit vide, soit l'option || EVENT (s) ||.

L'instruction éliminant un enregistrement d'un fichier déclaré avec le nom x et les descripteurs || DIRECT || UPDATE || a la forme

|| DELETE FILE (x) FROM (y) KEY FROM (z) t; ||,

où t est soit vide, soit le complément || EVENT (s) ||.

L'instruction d'entrée à partir d'un fichier déclaré avec le nom x et les descripteurs || DIRECT || UPDATE || EXCLUSIVE || a la forme

|| READ FILE (x) INTO (y) KEY (z) ts; ||,

où t est soit vide, soit l'option || NOLOCK ||; s est soit vide, soit l'option || EVENT (q) ||.

Chaque instruction d'entrée à partir d'un fichier déclaré avec le descripteur || EXCLUSIVE || implique un blocage de l'enregistrement, si l'option || NOLOCK || n'est pas spécifiée. Un enregistrement bloqué par une branche ne peut être lu, éliminé ou recopié par aucune autre branche, jusqu'à ce qu'il ne soit débloqué. Toute tentative de lire, d'effacer, de recopier ou de bloquer un enregistrement par une branche autre, que celle qui a opéré le blocage amène à une attente. Le déblocage peut être effectué par la branche qui a bloqué au moyen de l'une des instructions suivantes: || UNLOCK ||;

|| READ || avec l'option || NOLOCK||; || REWRITE|| ou || DELETE||; || CLOSE||.

L'instruction qui débloque un enregistrement sur un fichier déclaré avec le nom x et les descripteurs || DIRECT|| UPDATE|| EXCLUSIVE|| a la forme

|| UNLOCK FILE (x) KEY (y);||.

§ 9.8. Moyens d'interruption

Le langage PL/1 dispose des moyens qui assurent l'interruption de la marche normale du programme dans certaines conditions appelées *situations d'interruption*. Les moyens d'interruption sont les situations d'interruption et les réactions standard aux situations d'interruption, les instructions d'interruption || ON|| REVERT|| SIGNAL||DISPLAY||.

9.8.1. Situations d'interruption. Les situations d'interruption sont groupées comme suit: situations de calcul, situations d'entrée-sortie, situations de contrôle du programme, situation de traitement de listes, situation définie par le programmeur, situation de réaction du système.

Pour les situations de calcul et les situations de contrôle du programme, le nom de la situation (l'éventualité de la situation) est introduit par le programmeur sous d'option dans l'instruction correspondante (voir le § 9.7). Les autres situations sont présentes implicitement.

Situations de calcul. Ce sont les situations || CONVERSION|| FIXED OVER FLOW|| OVER FLOW|| UNDER FLOW|| ZERODIVIDE||.

La *situation* || CONVERSION|| se produit lors d'une conversion d'une chaîne de caractères distincts de 0 et 1 en une chaîne binaire, lorsqu'apparaissent des symboles inadmissibles, le résultat visé étant du type arithmétique (voir les pp. 9.5.1, 9.5.3).

Réaction du système: la signalisation et la production de la situation || ERROR||.

La *situation* || FIXEDOVER FLOW|| se produit lors des opérations arithmétiques sur les nombres en point fixe si le résultat dépasse les limites du champ numérique. Le résultat est alors tronqué.

Réaction du système: la signalisation et la poursuite des calculs.

La *situation* || OVER FLOW|| se produit lors des opérations arithmétiques sur les nombres en point flottant si l'exposant du résultat dépasse sa valeur maximale admissible.

Réaction du système: la signalisation et la production de la situation || ERROR||.

A la *situation* || UNDER FLOW || amènent les opérations arithmétiques sur les nombres en point flottant si l'exposant du résultat est inférieur à sa valeur minimale admissible. Le zéro est fourni en tant que résultat. La situation ne se produit pas lorsque les opérandes d'une soustraction sont deux nombres égaux.

Réaction à l'interruption: la signalisation et la poursuite des calculs.

La *situation* || ZERODIVIDE || se produit lorsqu'une division par le zéro est effectuée. Le résultat est indéterminé.

Réaction à l'interruption: la signalisation et la production de la situation || ERROR ||.

Les situations d'entrée-sortie. Ce sont les situations || ENDFILE || END PAGE || TRANSMIT || UNDEFINEDFILE || NAME || KEY || RECORD ||.

La situation || END FILE || se produit au cours de l'exécution de l'une des instructions || GET || READ || si l'on essaie de lire dans un fichier l'information qui se trouve en dehors du limiteur du fichier.

Le système ayant réagi à l'interruption, l'exécution du programme est reprise à partir de l'instruction suivante.

Réaction du système: la signalisation et la production de la situation || ERROR ||.

La *situation* || END PAGE || se produit lors de l'exécution d'une instruction de sortie || PUT || orientée vers un flux de nom x, lorsqu'il est essayé de commencer une nouvelle ligne en dehors des bornes déterminées par l'option || PAGE SIZE || (voir le p. 9.7.18.9) spécifiée dans l'instruction || DECLARE || (voir le p. 9.7.17).

Réaction du système: une nouvelle page commence.

La *situation* || TRANSMIT (x) || se produit lors de l'exécution d'instructions d'entrée-sortie, s'il y a erreur de transmission de données sur un fichier de nom x.

Ayant réagi à l'interruption, le système continue l'exécution du porgramme comme s'il n'y avait pas d'erreur.

Réaction à l'interruption: la signalisation et la production de la situation || ERROR ||.

La *situation* || UNDEFINEDFILE (x) || se produit lors de l'exécution d'une instruction || OPEN || si les options spécifiées pour le fichier x sont contradictoires.

Réaction du système: la signalisation et la production de la situation || ERROR ||.

La *situation* || NAME (x) || se produit lors de l'exécution d'une instruction || GET || qui lit dans un fichier x représentant un flux commandé par les données (voir les pp. 9.7.18.4, 9.7.18.8), si l'affectation introduite contient un identificateur ne figurant pas dans la liste d'entrée-sortie (voir le p. 9.7.18.3).

Ayant réagi à l'interruption, le système poursuit l'exécution de l'instruction || GET||.

Réaction du système: la signalisation et la non-considération de l'affectation introduite.

La *situation* || KEY (x)|| se produit lorsque: a) au cours de l'exécution de l'une des instructions || READ||REWRITE|| DELETE|| (voir le p. 9.7.18.10), le mot-clé n'est pas trouvé; b) au cours de l'exécution de l'instruction || WRITE|| ou || LOCATE||, le mot-clé a déjà existé.

Après l'exécution de la réaction à l'interruption, le programme s'exécute comme s'il n'y avait pas d'erreur.

Réaction à l'interruption: la signalisation et la production de la situation || ERROR||.

La *situation* || RECORD (x)|| se produit lors de l'exécution d'instruction || READ|| ou || REWRITE|| si la taille de l'enregistrement diffère de celle de la variable. Les actions suivantes sont alors accomplies: a) si l'enregistrement dépasse la variable, les données superflues de l'enregistrement sont perdues; b) si la variable dépasse l'enregistrement, les données en excès de la variable ne sont pas transmises en sortie et ne sont pas modifiées en entrée.

Réaction à l'interruption: la signalisation et la production de la situation || ERROR||.

Les situations de contrôle du programme. Ce sont les situations || SUBSCRIPTRANGE|| CHECK||.

La *situation* || SUBSCRIPTRANGE|| se produit lorsque la valeur calculée d'un indice sort de ces limites.

Réaction du système: la signalisation et la production de la situation || ERROR||.

La *situation* || CHECK ($x_1, \ldots, x_n$)|| se produit lorsque 1) il y a appel d'une instruction ayant pour étiquette un élément de la liste ($x_1, \ldots, x_n$); 2) il y a appel d'une procédure par le nom d'entrée qui coïncide avec un élément de la liste ($x_1, \ldots, x_n$); 3) s'exécute une instruction d'affectation dont la partie gauche coïncide avec un nom de variable simple (de tableau, de structure) figurant sur la liste ($x_1, \ldots, x_n$); 4) dans l'instruction || DO||, en tant que variable de contrôle est utilisée une variable dont le nom figure sur la liste ($x_1, \ldots, x_n$); 5) dans l'instruction || GET||, l'un au moins des éléments de la liste coïncide avec un élément de la liste ($x_1, \ldots$ $\ldots, x_n$); 6) dans l'option || SET|| d'une instruction || ALLOCATE||, || READ|| ou || LOCATE|| est utilisée une variable dont le nom figure sur la liste ($x_1, \ldots, x_n$); 7) dans l'instruction || GO TO|| ou || RETURN|| est utilisée une variable dont le nom figure sur la liste ($x_1, \ldots, x_n$).

Après la réalisation de la situation, l'exécution du programme continue.

Réaction du système: si l'identificateur représente une étiquette

d'instruction, un nom d'entrée, une variable du type étiquette, un nom de branche ou d'événement, alors cet identificateur est imprimé sur le fichier de mise au point standard; sinon il est également imprimé sur un fichier de mise au point mais conformément au format défini pour la sortie commandée par les données.

La situation de traitement de listes. Cette situation appelée || AREA|| se produit si l'on essaie de stocker une variable dans un domaine dont les dimensions ne le permettent pas.

Réaction du système: la production de la situation || ERROR||.

La situation définie par le programmeur. Elle a la forme || CONDITION (x)||, où x est un identificateur extérieur donné par le programmeur. La situation se produit à la suite d'exécution d'une instruction || SIGNAL|| avec l'identificateur x.

Réaction du système: la signalisation et la poursuite de l'exécution du programme.

Les situations de réaction du système. Ce sont les situations || FINISH|| et || ERROR||.

La *situation* || FINISH|| se produit immédiatement avant l'arrêt de l'exécution de la procédure principale par les instructions || STOP|| RETURN|| END||. Si la réaction à l'interruption causée par cette situation est indiquée, les actions correspondantes s'exécutent comme une partie de la branche dans laquelle l'interruption s'est produite. Après la réaction à l'interruption, le système termine l'exécution de la branche principale.

La *situation* || ERROR|| se produit lorsque l'exécution de la branche principale doit être arrêtée à cause d'une erreur. Si dans ce cas une réaction à l'interruption est prévue, l'exécution de cette réaction est suivie de la situation || FINISH||.

9.8.2. Instruction donnant une réaction à l'interruption. L instruction détermine la réaction à l'interruption causée par une situation indiquée dans cette instruction. Elle a l'une des formes suivantes:

|| ON *abc*||,

|| ON *ab* SISTEM;||,

où *a* est l'une des situations d'interruption admises par le système (voir le p. 9.8.1); *b* est soit vide, soit l'option || SNAP||; *c* est soit vide, soit une réaction à l'interruption définie par le programmeur. Elle peut être ou bien une instruction non étiquetée différente des instructions || BEGIN|| DO|| END|| RETURN|| FORMAT|| PROCEDURE|| DECLARE||, ou bien un bloc non étiqueté et ne contenant pas l'instruction || RETURN||. Si *c* est vide, aucune réaction à l'interruption ne se produit.

Si a est une || CONDITION ||, alors l'interruption ne peut avoir lieu qu'au cours de l'exécution de l'instruction || SIGNAL || définissant cette situation.

La première forme de l'instruction || ON || permet au programmeur de prévoir une réaction voulue à l'interruption, la deuxième forme est orientée vers les réactions standard du système (voir le p. 9.8.1).

L'option || SNAP || indique qu'il faut fournir une information sur l'état du programme au moment d'interruption.

Une réaction à l'interruption spécifiée dans l'instruction || ON || reste en vigueur dans le bloc qui contient cette instruction || ON ||, ainsi que dans tous les blocs qui lui sont dynamiquement subordonnés et cela jusqu'à ce qu'une nouvelle instruction || ON || dans le même bloc ne soit exécutée. Et si une nouvelle instruction est exécutée dans un bloc dynamiquement subordonné au bloc donné, alors la réaction à l'interruption définie par l'ancienne instruction || ON || est provisoirement annulée et ne sera restituée qu'à la suite d'exécution d'une instruction || REVERT || (voir le p. 9.8.3) et après la fin du fonctionnement du bloc contenant la nouvelle instruction || ON ||.

La réaction à l'interruption définie par l'instruction || ON || s'étend au domaine d'action de la procédure ou du bloc contenant cette instruction.

9.8.3. Instruction annulant la réaction à l'interruption. Cette instruction a la forme

|| REVERT a ; ||,

où a est l'une des situations d'interruption prévues dans le système (voir le p. 9.8.1). Elle est destinée à annuler la réaction à l'interruption définie par une instruction || ON || conformément à la situation spécifiée dans cette instruction || ON ||.

La situation spécifiée dans l'instruction || REVERT || coïncide avec celle de l'instruction || ON ||, ces deux instructions étant intérieures à un même bloc (à une même procédure).

L'exécution de l'instruction || REVERT || restitue la réaction à l'interruption établie dans le bloc (la procédure) dans lequel le bloc en question est dynamiquement emboîté.

9.8.4. Instruction imitant la situation d'interruption. Cette instruction a la forme

|| SIGNAL a ; ||,

où a est l'une des situations d'interruption prévues dans le système (voir le p. 9.8.1).

28-0779

L'exécution de cette instruction imite la réalisation de la situation a. Ceci fait, le contrôle passe au programme de réaction à l'interruption donné dans l'instruction || ON || (voir le p. 9.8.2) où est spécifiée la situation a et qui est exécutée vers le moment d'exécution de l'instruction donnée || SIGNAL ||. Après l'exécution du programme de réaction à l'interruption le contrôle passe à l'instruction qui suit immédiatement l'instruction || SIGNAL a; ||.

Si la situation a indiquée dans l'instruction || SIGNAL || n'est pas initialisée (i.e. vers le moment d'exécution de l'instruction || SIGNAL a; || l'instruction || ON || contenant la situation a n'est pas exécutée ou n'existe pas dans le programme), l'interruption n'a pas lieu, donc l'instruction || SIGNAL a; || est équivalente à une instruction vide.

9.8.5. Instruction de sortie sur display. Cette instruction a la forme

|| DISPLAY (w) xy; ||,

où w est une expression scalaire; x est soit vide, soit l'option || REPLY (z) || où z est une variable du type chaîne; y est soit vide, soit l'option || EVENT (t) || dans laquelle t est une variable du type événement.

Au cours de l'exécution de l'instruction || DISPLAY || un message est fourni au programmeur; c'est une chaîne de caractères qui représente le résultat du calcul de la valeur de l'expression w éventuellement convertie en une chaîne de caractères de longueur maximale (si x et y sont vides). Après l'exécution de cette instruction le contrôle passe à l'instruction suivante.

Si x est non vide et y est vide, on attribue à la variable z la valeur de w qui est fournie comme message, après quoi le programme s'arrête.

Si x et y sont non vides, le programme ne s'arrête pas. La variable t prend la valeur || '0'B ||; s'il y a une réaction du programmeur, la valeur de la variable t devient || '1'B ||.

§ 9.9. Fonctions incorporées

Les fonctions les plus usitées sont considérées dans le PL/1 comme éléments du langage. Elles s'appellent *fonctions incorporées*. Parmi ces fonctions il y a certaines fonctions mathématiques, des fonctions arithmétiques, des fonctions pour le traitement de chaînes, des fonctions pour le traitement de tableaux.

On considère comme *fonctions mathématiques* les fonctions circu-

laires et hyperboliques, l'extraction de la racine carrée, l'élévation à une puissance, les fonctions logarithmiques. Les arguments de ces fonctions doivent être donnés sous forme arithmétique en point flottant. Dans le cas contraire, avant d'appeler la fonction, ses arguments sont mis sous cette forme d'après les règles formulées en 9.6.1. La valeur fournie est un nombre en point flottant, ayant la base et la précision de l'argument.

Les arguments de fonctions mathématiques peuvent être des expressions scalaires ou des tableaux. Dans le cas d'un tableau, la valeur fournie par la fonction représente un tableau de même dimension et de mêmes bornes que les arguments; la fonction s'exerce sur chaque élément d'un tableau-argument.

Les caractéristiques principales des fonctions mathématiques sont données dans la table 9.2.

Fonctions arithmétiques. Les arguments de fonctions arithmétiques sont représentés sous forme arithmétique avec les descripteurs || FLOAT || FIXED || et || DECIMAL || BINARY ||. En l'absence de ces descripteurs avant d'appeler la fonction on ramène l'argument à la forme arithmétique par omission. Une expression scalaire ou un tableau peuvent servir d'argument. Dans le dernier cas, la valeur fournie par la fonction est un tableau de même dimension et de mêmes bornes que les arguments. Chaque fonction fournit une valeur sous forme arithmétique. Si la base, la forme de représentation, la précision de cette valeur ne sont pas indiquées, elles sont les mêmes que pour les arguments.

La table 9.3. contient les caractéristiques principales des fonctions arithmétiques incorporées. On y utilise les désignations suivantes: m-le nombre maximal de positions, p-le nombre total de positions dans la représentation d'un nombre, q-le nombre de positions après le point décimal (binaire) dans la représentation d'un nombre (m, p et q sont des entiers décimaux; m et p sont positifs, q peut être négatif).

Les *fonctions de traitement des chaînes* s'utilisent dans les opérations sur les chaînes. Si un argument n'est pas une chaîne, on le transforme, avant d'appeler la fonction, en une chaîne binaire ou une chaîne de caractères.

La table 9.4. contient les caractéristiques principales des fonctions de traitement des chaînes.

Les *fonctions de traitement des tableaux* (table 9.5) ont pour arguments les tableaux de scalaires. Les valeurs fournies sont des scalaires.

Autres fonctions incorporées. Ce sont les fonctions qui permettent, par exemple, de savoir l'heure et la date au cours de l'exécution du programme, ou bien de calculer des adresses courantes de la mémoire de variables, etc. La table 9.6 contient les caractéristiques principales de ces fonctions.

Table 9.2

Caractéristiques principales des fonctions mathématiques

<table>
<tr><th>Fonction</th><th>Valeur de la fonction</th><th>Remarque</th></tr>
<tr><td>|| EXP (x) ||
|| LOG (x) ||
|| LOG 2 (x) ||
|| LOG 10 (x) ||</td><td>exp (x)
$\log_e (x)$
$\log_2 (x)$
$\log_{10} (x)$</td><td>La situation || ERROR || se produit pour $x \leqslant 0$</td></tr>
<tr><td>|| ATAN (x) ||
|| TAN (x) ||
|| SIN (x) ||
|| COS (x) ||</td><td>arc tan (x)
tan (x)
sin (x)
cos (x)</td><td>L'argument x est représenté en radians. Si le nom de fonction est complété à droite par le symbole || D || (ATAND (x), TAND (x), etc.) cela signifie que l'argument est donné en degrés</td></tr>
<tr><td>|| ATAN (y, x) ||</td><td>La valeur de la fonction vaut : arc tan (y/x) pour $x > 0$; $\pi/2$ pour $x=0$, $y>0$; $-\pi/2$ pour $x=0$, $y < 0$; $\pi+$arc tan (y/x) pour $x < 0$, $y \geqslant 0$; $-\pi+$arc tan (y/x) pour $x<0$, $y < 0$.</td><td>L'argument se transforme de façon à avoir les caractéristiques prioritaires correspondantes de y et x. La situatton || ERROR || se produit pour $x=0$, $y=0$</td></tr>
<tr><td>|| TANH (x) ||
|| COSH (π) ||
|| SINH (x) ||</td><td>tanh (x)
cosh (x)
sinh (x)</td><td>L'argument x est représenté en radians</td></tr>
<tr><td>|| ATANH (x) ||
|| ATAND (x, y) ||
||ERF (x) ||</td><td>arc tanh (x)
$(180/\pi) *$ ATAN (y, x)
$(2/\sqrt{\pi} \int_0^x \text{EXP}(-t^2)\, dt$</td><td>La situation || ERROR || se produit pour ABS (x) $\geqslant 1$</td></tr>
<tr><td>||SQRT (x) ||</td><td>$+x^{1/2}$</td><td>La situation || ERROR || se produit pour $x < 0$</td></tr>
</table>

Table 9.3

Caractéristiques principales des fonctions arithmétiques

Fonction	Valeur de la fonction
‖ ABS (x) ‖	$\|x\|$
‖ BINARY (x, p, q) ‖	Valeur binaire de x avec la précision (p, q). Si x est du type ‖ FLOAT ‖, on peut ne pas donner q
‖ CELL (x) ‖	Le plus petit entier supérieur ou égal à x (borne supérieure). Si x est un nombre en point fixe la précision du résultat est (min $(m$, max $(p+1-q, 1)$ 0)
‖ DECIMAL (x, p, q) ‖	Valeur décimale de x avec la précision (p, q). Si x est du type ‖ FLOAT ‖, on peut ne pas donner q
‖ FIXED (x, p, q) ‖	Valeur de x en point fixe avec la précision (p, q). Si $q=0$, on peut ne pas le donner
‖ FLOAT (x, p) ‖	Valeur de x en point flottant avec la précision p
‖ FLOOR (x) ‖	Le plus grand entier inférieur ou égal à x (borne inférieure). Si x est un nombre en point fixe, la précision du résultat est min $(m$, max $(p+1-q, 1), 0)$
‖ MAX $(x_1, \ldots, x_n)$ ‖ $(n \geqslant 2)$	Valeur de l'argument maximal avec les caractéristiques de l'argument le plus long. Si tous les arguments sont en point fixe et de précisions respectives $(p_1 q_1)$, $(p_2, q_2), \ldots, (p_n, q_n)$, la valeur de la fonction aura la précision (min $(m$, max $(p_1-q_1, \ldots, p_n-q_n)+$max $(q_1, \ldots, q_n))$, max $(q_1, \ldots, q_n))$
‖ MIN $(x_1, \ldots, x_n)$ ‖ $(n \geqslant 2)$	Valeur de l'argument minimal avec les caractéristiques de l'argument le plus long. Si tous les arguments sont en point fixe et de précisions respectives $(p_1\ q_1)$, $(p_2, q_2), \ldots, (p_n, q_n)$, la valeur de la fonction aura la précision (min $(m$, max $(p_1-q_1, p_2-q_2, \ldots, p_n-q_n)+$max $(q_1 \ldots, q_n)$, max $(q_1, \ldots, q_n))$

Suite de la table 9.3

Fonction	Valeur de la fonction
|| MOD (x, y) || La base de numération et la forme de représentation sont choisies égales aux caractéristiques prioritaires des deux arguments	La valeur de la fonction est égale au reste positif de la division de x par y. Pour les nombres en point fixe, la précision du résultat est $(\min(m,\ p_y - q_y + \max(p_x,\ q_x)),\ \max(q_x,\ q_y))$
|| PRECISION (x, p, q) || Si x est un nombre en point fixe, il faut donner q, sinon on peut l'omettre	La valeur de la fonction est égale à x avec la précision (p, q)
|| TRUNS (x) ||	La valeur de la fonction est celle de || CELL (x) || pour $x < 0$, et celle de || FLOOR (x) || pour $x \geqslant 0$. Si x est un nombre en point fixe précision (p, q), la précision du résultat est $\min(m, \max(p-q+1,1), 0)$
|| COMPLEX (x, y) ||, où x et y sont des nombres réels	La valeur de la fonction est le nombre complexe $x + iy$. La base, la forme de représentation et la précision correspondent aux caractéristiques prioritaires des arguments x et y
|| REAL (x) ||, où x est un nombre complexe	La valeur de la fonction est la partie réelle de l'argument avec toutes ses caractéristiques
|| IMAG (x) ||, où x est un nombre complexe	La valeur de la fonction est la partie imaginaire de l'argument avec toutes ses caractéristiques

Table 9.4

Caractéristiques principales des fonctions de traitement des chaînes

Fonction	Valeur de la fonction
‖ BIT (x, l) ‖, où x est une chaîne de caractères ou binaire ou une expression arithmétique ; l est un entier décimal non signé	Chaîne binaire de longueur l en laquelle se transforme la valeur de x. On peut ne pas donner l, alors la longueur du résultat sera déterminée par celle de l'argument
‖ BOOL (x, y, w) ‖, où x et y sont des constantes binaires ou chaînes de caractères ; w est la fonction de transformation représentant une chaîne binaire de 4 bits : $w_1\, w_2\, w_3\, w_4$	Chaîne binaire z dont la longueur est égale à la plus grande des longueurs de x et y. Si x et y sont de longueurs différentes, la chaîne plus courte est complétée à droite par des zéros. Les bits z_i de la chaîne z sont calculés selon les règles suivantes : $$z_i = \begin{cases} w_1 \text{ si } x_i=0 \text{ et } y_i=0; \\ w_2 \text{ si } x_i=0 \text{ et } y_i=1; \\ w_3 \text{ si } x_i=1 \text{ et } y_i=0; \\ w_4 \text{ si } x_i=1 \text{ et } y_i=1. \end{cases}$$
	Exemple. Le résultat de la fonction ‖ BOOL (X, Y, W) ‖, où X = = ‖ '11001'B ‖, Y = ‖ '01100'B ‖ et W = ‖ '1001'B ‖ est la chaîne binaire Z = ‖ '01010'B ‖
‖ CHAR (x, l) ‖, où x est une chaîne binaire, une chaîne de caractères ou une expression arithmétique, l est un entier décimal positif	Chaîne de caractères de longueur l en laquelle se transforme la valeur de x d'après les règles du § 9.7. La donnée de l n'est pas obligatoire. Si l'on ne la spécifie pas, la longueur de la chaîne transformée coïncide avec celle de l'argument
‖ HIGH (l) ‖, où l est un entier décimal non signé	Chaîne de caractères FF...F de longueur l

Suite de la table 9.4

Fonction	Valeur de la fonction
|| INDEX (x, y) ||, où chacun des arguments x et y est une chaîne binaire ou une chaîne binaire ou une chaîne de caractères	Entier binaire non signé qui indique le numéro de la position dans la chaîne x (à compter de gauche à droite) par laquelle commence la chaîne y ; zéro si y ne figure pas dans x
|| LOW (l) ||, où l est un entier décimal non signé	Chaîne de caractères de longueur l dont chaque caractère est un nombre hexadécimal 0
|| REPEAT (x, w) ||, où x est une chaîne binaire ou une chaîne de caractères, w est un entier décimal non signé	Chaîne obtenue par concaténation des chaînes x prises w fois
|| SUBSTR (x, p, l) ||, où x est une chaîne binaire ou une chaîne de caractères de longueur k, p est une expression scalaire convertie en un entier i, l est un entier décimal non signé, x peut être un tableau de chaînes, alors p est aussi un tableau	Chaîne vide si $i > k$, chaîne de longueur n qui représente une sous-chaîne de x ayant pour premier symbole le m-ème symbole de la chaîne x. Ici $m = \max(i, l)$. On peut ne pas donner l. Alors $n = k - m + 1$. Si x est un tableau de chaînes, le résultat représente aussi un tableau dont chaque élément est calculé à partir de l'élément correspondant du tableau x selon les règles décrites plus haut
|| UNSPEC (x) ||, où x est une chaîne de caractères ou une variable scalaire du type arithmétique, du type chaîne ou pointeur	Chaîne binaire qui représente x dans le code machine interne, de longueur égale (en bits) à celle de représentation interne de l'argument

Table 9.5

Caractéristiques principales des fonctions pour le traitement des tableaux

Fonction	Valeur de la fonction
‖SUM (x)‖, où x est du type ‖FLOAT‖ sinon les éléments du tableau x se transforment en ce type	Somme scalaire de tous les éléments du tableau x représentée en point flottant. La base de numération et la précision coïncident avec les caractéristiques respectives des éléments du tableau x
‖PROD (x)‖, où x est du type ‖FLOAT‖ sinon les éléments du tableau x se transforment en ce type	Scalaire égal au produit de tous les éléments du tableau x et représenté en point flottant. La base de numération et la précision coïncident avec celles des éléments du tableau x
‖ALL (x)‖, où les éléments du tableau x sont des chaînes binaires, sinon ils se transforment en ce type	Chaîne binaire de longueur égale à celle des éléments du tableau x et dont le i-ème bit vaut 1 si les i-èmes bits de tous les éléments du tableau x le sont, dans le cas contraire le i-ème bit est nul
‖ANY (x)‖, où les éléments du tableau x sont des chaînes binaires, sinon ils se transforment en ce type	Chaîne binaire de longueur égale à celle des éléments du tableau x et dont le i-ème bit vaut 0 si les i-èmes bits de tous les éléments du tableau x valent 0, dans le cas contraire le i-ème bit du résultal vaut 1
‖LBOUND (x, s)‖ (‖HBOUND (x, s)‖), où s est une expression scalaire qui est convertie en entier binaire n	Entier égal à la borne inférieure (supérieure) courante du tableau x relative à la n-ème dimension. Remarque. Ces fonctions ne sont pas définies si l'argument x n'est pas implanté en mémoire, ou si la dimensionnalité de x est inférieure à n, ou si $n \leqslant 0$

Table 9.6

Caractéristiques principales des fonctions spéciales

Fonction	Valeur de la fonction
‖ DATE ‖	Chaîne de caractères ayant la structure $x_1x_2y_1y_2z_1z_2$, où x_1x_2 sont deux derniers chiffres de l'année courante, y_1y_2 est un nombre déterminant le mois, z_1z_2 un nombre déterminant la date
‖ TIME ‖	Chaîne de caractères indiquant l'heure courante. La chaîne a la forme $x_1x_2y_1y_2z_1z_2t_1t_2t_3$, où x_1x_2 détermine les heures, y_1y_2 les minutes, z_1z_2 les secondes, $t_1t_2t_3$ les millisecondes
‖ STRING (x) ‖, où x est une structure empaquetée, composée soit de chaînes binaires et de champs numériques binaires, soit de chaînes de caractères et de champs numériques décimaux	Chaîne obtenue par concaténation de tous les éléments de la structure x
‖ EVENT (x) ‖, où x est un nom de variable scalaire du type événement	Chaîne binaire ‖ '0'B ‖ ou ‖ '1'B ‖ selon l'état courant de l'événement de nom x (voir le p. 9.6.5)
‖ ADDR (x) ‖, où x est un nom de variable	Valeur scalaire du pointeur qui indique l'adresse de la variable x dans la mémoire
‖ NULL ‖	Cette fonction détermine seule la valeur nulle du pointeur, donc ne définit aucune génération de données
‖ PRIORITY (x) ‖, où x est un nom de variable du type branche	La branche de nom x devient prioritaire par rapport à la branche dans laquelle la fonction donnée s'exécute
‖ ALLOCATION (x) ‖, où x est une variable non basée et non contrôlée.	‖ '1'B ‖ si x est implanté en mémoire, ‖ '0'B ‖ dans le cas contraire (voir l'exemple 9.35)

CHAPITRE 10

INTRODUCTION AU LANGAGE COBOL

Au moment de l'élaboration du langage algorithmique COBOL, l'ALGOL était déjà connu et le FORTRAN avait été mis en pratique avec succès. La première description du COBOL a paru en 1961, après un essai de ce langage s'étendant sur deux années environ, qui a été réalisé par un Comité du département de la défense des U.S.A., avec la participation de représentants des plus grandes firmes fabriquant et exploitant les ordinateurs.

Le COBOL (Common Business Oriented Language) est un langage algorithmique orienté vers une classe de problèmes; il est destiné à décrire certains processus de traitement des données.

En élaborant le COBOL, ses auteurs n'ont pas oublié que la possibilité d'appeler un programme à partir d'un autre programme est un grand avantage. Dans ce but il a fallu imposer certaines conditions à la structure de programmes et à leur présentation. En particulier, l'une des mesures possibles consiste à munir chaque programme d'un nom de format standard et à définir certains paramètres du programme appelé.

Les constructeurs du COBOL ont prévu la possibilité d'utiliser des calculateurs différents pour la traduction et pour le traitement du programme compilé. En outre, ils ont admis qu'un même programme COBOL puisse être traduit dans les différents langages de machine.

Pour les problèmes de traitement de fichiers, les données initiales et les résultats se présentent sous la forme de diverses tables dont la structure exerce une influence non négligeable sur la forme du programme.

Il est à signaler que, dans le COBOL, on a essayé d'assurer l'indépendance de la précision des calculs des performances techniques de la machine (en utilisant le cadrage de données). Pourtant, il faut reconnaître que ce problème n'a pas été complètement résolu car, en principe (lorsque les calculs sont très longs ou de grandes erreurs sont inévitables), les grandeurs cadrées peuvent contenir une erreur (dépendant de la capacité des registres de l'unité de calcul et des règles d'arrondissement adoptées).

Il faut dire que le COBOL représente une tentative, d'ailleurs réussie, d'utiliser pour la programmation une langue naturelle (l'anglais en l'occurrence). De toute façon, quiconque possède des connaissances minimales sur l'organisation des calculateurs et sur la programmation pourra sinon programmer en COBOL, au moins comprendre un programme COBOL.

Dans notre exposé du COBOL nous avons omis quelques détails et suivi le même schéma que nous avons adopté pour les autres langages décrits dans ce livre. Cela permet de mieux voir ce qu'il y a de commun et de spécifique dans les langages. Comme pour les autres langages algorithmiques, nous nous servons du double trait vertical || lorsqu'il faut séparer des textes COBOL. Pour fixer les idées, nous exposons le COBOL IBM-360.

§ 10.1. Alphabet du langage COBOL

Les symboles de base du COBOL, qui forment son alphabet, sont:

1) les *chiffres* || 0|| 1|| 2|| 3|| 4|| 5|| 6|| 7|| 8|| 9||;

2) les *lettres majuscules latines* de A à Z;

3) les *limiteurs*: les *parenthèses gauche* || (|| *et droite* ||) || et l'*apostrophe* || ' ||;

4) les *séparateurs* || ␣ || . || , || ; || qui s'appellent respectivement espace *), ou blanc, point, virgule, point-virgule;

5) les *signes d'opérations arithmétiques* || + || — || * || / || appelés respectivement plus, moins, signe de multiplication et signe de division;

6) les *signes de relations* || >|| <|| = || qui se lisent respectivement « est supérieur à », « est inférieur à », « est égal à ».

Remarquons que le signe || — || est utilisé non seulement comme signe d'opération arithmétique, mais aussi comme tiret. Le signe d'égalité joue encore le rôle de signe d'affectation. L'exponentiation est exprimée par le symbole || ** ||.

§ 10.2. Constructions primaires du langage COBOL

Les constructions primaires du COBOL sont les *mots* COBOL. On appelle mot toute suite non vide de symboles de base, sans espaces ni apostrophes ou toute suite de cette forme entre apostrophes.

EXEMPLE 10.1. Les constructions || TABLE1 || || 'TABLE1' || SITUATION-DE-FAMILLE || +125 || —10.12 || X++—Y5 || sont des mots. Les textes || PREMIER ␣ VALUE || PREMIER VALUE || ne sont pas des mots puisque contiennent des espaces.

---

* Dans ce que suit, un espace sera également figuré comme un espace entre deux symboles.

Toutes les autres constructions du COBOL sont formées à partir de mots et de symboles isolés. Les frontières entre les mots ou les symboles isolés sont interprétées par des espaces *). Dans le COBOL, contrairement à l'ALGOL par exemple, les espaces ont une signification indépendante. Cela permet de faire la distinction entre un tiret (qui n'est pas encadré d'espaces) et un signe moins (il se trouve entre deux espaces).

Parmi tous les mots, dégageons tout d'abord la classe des mots alphanumériques.

On appelle *mot alphanumérique* soit un mot formé uniquement par des chiffres et lettres (appartenant aux symboles de base), soit un mot obtenu en liant deux mots alphanumériques par un tiret. En vertu de cette définition récursive, un mot alphanumérique peut se composer de plusieurs « fragments » liés par des tirets, chaque « fragment » ne contenant que des chiffres et des lettres. Un mot alphanumérique ne peut pas commencer ni se terminer par un tiret.

Chaque symbole || . || , || ; ||, s'il termine un mot alphanumérique, n'en est pas séparé par un espace, mais lui-même doit être suivi d'un espace. Les symboles || (||) || séparent les mots alphanumériques sans l'emploi d'espaces.

Il est convenu dans le COBOL que la longueur d'un mot alphanumérique ne dépasse pas 30 caractères. En plus des mots alphanumériques, on utilise en COBOL des mots qui peuvent contenir outre les chiffres, lettres et tirets autres symboles de base.

Tous les mots du vocabulaire du COBOL sont partagés en deux groupes: mots réservés et mots non réservés.

10.2.1. Mots réservés. On classe dans ce groupe les mots alphanumériques inclus dans la *liste des mots réservés*. Ce sont des mots d'une langue naturelle qui doivent être écrits correctement, d'après les règles adoptées dans le COBOL. Leur utilisation est analogue à celle des symboles de base-mots en ALGOL. On leur attribue une signification particulière. Ces mots et leur emploi sont décrits plus bas, dans la section consacrée à la syntaxe du COBOL.

10.2.2. Mots non réservés. Ce sont des mots qu'on ne trouve pas sur la liste des mots réservés. Les différents groupes de mots non réservés diffèrent l'un de l'autre par la sémantique, bien que cela n'exclue pas de différences syntaxiques.

Les *noms externes* sont les mots alphanumériques commençant chacun par une lettre et contenant de un à huit caractères.

On les utilise pour nommer des programmes et des collections de données en entrée-sortie.

---

*) C'est une question technique: la méthode de perforation en dépend. D'autres règles de séparation de mots sont possibles dans d'autres versions du COBOL.

Exemple 10.2. || PROGR-21 || D123-A2 || PROG-12 || CALCUL || sont des noms externes. Les mots || 12 B 41 || 8625 || ne le sont pas.

Un *nom de donnée* est un mot alphanumérique contenant au moins une lettre. Ces mots n'ont pas de significations fixes et s'utilisent pour désigner les opérandes (données, indices), les procédures et les organes du calculateur. Leur longueur ne dépasse en général pas 30 caractères.

Exemple 10.3. Les mots || 18235 || ou || 182-35 || ne peuvent pas servir de noms de donnée. On peut utiliser comme tels les mots || A12583 || DATE-ACHAT || 1-A25 || 125-TAUX ||.

Un *nom indicé* représente une notation qui se compose d'un nom de donnée suivi d'une parenthèse gauche, d'une liste d'indices et d'une parenthèse droite. La liste d'indices comprend un, deux ou trois indices. Dans les deux derniers cas les indices sont séparés par des virgules. Un indice représente un nom de donnée ou un entier non signé.

Exemple 10.4. Les notations suivantes peuvent servir de noms indicés: || ENTREPRISE (2, 3, 5) || LIGNE-DE-TABLE (41) || LIGNE-DE-TABLE (NUMERO-DE-LIGNE) || FRAIS-TRANSPORT (CLIENT-DE-POT) ||.

Un *littéral* est ou bien un littéral non numérique, ou bien un littéral numérique, ou bien une constante figurative.

On appelle *littéral non numérique* un mot formé de n'importe quels caractères du calculateur, sauf l'apostrophe, ayant une longueur ne dépassant pas 120 caractères et délimité par des apostrophes (la longueur d'un littéral avec les apostrophes peut donc atteindre 122 caractères).

Exemple 10.5. Les notations || '12—>=X' || 'TABLE — 1' || '1283562' || sont des littéraux non numériques.

Un *littéral numérique* est soit un littéral à point fixe, soit un littéral à point flottant.

On appelle *littéral contenant un point fixe* un mot formé uniquement des chiffres, ou un mot formé de chiffres précédés d'un point, ou un couple de mots formés de chiffres et séparés par un point (le deuxième mot du couple étant non vide), ou enfin un des mots ci-décrits précédé par un symbole || + || ou || — ||.

Le nombre de chiffres dans un littéral à point fixe ne doit pas dépasser 18.

EXEMPLE 10.6. Les notations suivantes sont des littéraux à point fixe :

|| 89573 || 4.625 || .32075 || .025 || +32187 || —3.624 || —.2125 ||.

On appelle *littéral à point flottant* une notation qui représente un littéral point fixe suivi du symbole || E || suivi d'un littéral point fixe sans point.

EXEMPLE 10.7. Les notations

|| +2.25E+3 || —7.25E+2 || .025E+3 || 125E0 || —.1225E+3 ||

sont des littéraux point flottant.

Les *constantes figuratives* sont des littéraux d'un type particulier qui nomment d'une manière standard certaines valeurs. Le singulier et le pluriel d'une constante figurative sont équivalents et s'emploient indifféremment. Voici des exemples de constantes figuratives :

|| ZERO || ou || ZEROS || ou || ZEROES || représente un ou plusieurs zéros ;

|| SPACE || ou || SPACES || représente un ou plusieurs blancs ;

|| QUOTE || ou || QUOTES || représente un ou plusieurs apostrophes ;

|| HIGH-VALUE || ou || HIGH-VALUES || représente un ou plusieurs caractères de rang le plus élevé dans la séquence de classement du calculateur ;

|| LOW-VALUE || ou || LOW-VALUES || représente un ou plusieurs caractères de rang le plus bas dans la séquence de classement du calculateur ;

|| ALL || suivi immédiatement (sans un blanc) d'un littéral non numérique représente le mot entre les apostrophes du littéral noté ou plusieurs fois de suite.

EXEMPLE 10.8. Voici quelques constantes figuratives de ce dernier type :

|| ALL'X' || ALL'5' || ALL' + ' || ALL'CP' ||.

Les littéraux (sauf les constantes figuratives) sont des entités qui, dans les programmes COBOL, peuvent servir de leurs propres noms. Les littéraux numériques représentent les nombres correspondants en virgule fixe ou en virgule flottante.

EXEMPLE 10.9. Dans l'expression arithmétique (voir le p. 10.8.1)

|| X + 2.5 ||

le mot à une lettre X est le nom d'une donnée, et le mot 2.5 est le littéral désignant le nombre 2,5. On pourrait donner à ce nombre le nom Y par exemple, et l'expression arithmétique prendrait alors la forme

|| X + Y ||.

Exemple 10.10. Lorsqu'il faut que le calculateur imprime le mot FIN, on peut inclure dans le programme COBOL correspondant l'instruction (voir le p. 10.12.3)

|| WRITE 'FIN' ||

On pourrait considérer les constantes figuratives comme des noms de données qui (les données) ont les valeurs établies une fois pour toutes, et ceci pour tous les programmes COBOL, et ne demandent donc pas d'être mentionnées dans la division des données (voir le § 10.6). Or, le fait que les valeurs des constantes figuratives sont fixes, comme celles des nombres, les rapproche des littéraux. Le nombre de symboles dans une suite représentée par une constante figurative est déterminé par la taille du champ réservé à cette suite.

Exemple 10.11. Si le format réservé à une donnée est à remplir par des zéros, on peut le faire à l'aide de l'instruction (voir les p. 10.13.1)

|| MOVE ZERO TO X ||.

S'il faut remplir le même format par des apostrophes ou des lettres, par exemple M, on peut écrire respectivement

|| MOVE QUOTE TO X || MOVE ALL'M' TO X ||.

En plus des mots non réservés énumérés, on peut employer dan la division des données d'un programme COBOL les mots de deux autres types: *numéro de niveau* et *chaîne* PICTURE. Un numéro de niveau est un entier non signé. Il est convenu qu'un numéro de niveau se compose de deux chiffres (dans certaines versions du langage, de un ou de deux). On utilise les numéros de niveaux pour décrire les données présentées sous la forme d'une table arborescente. On associe le niveau 01 à l'en-tête de la table; si un élément de la table est de niveau i et s'il embrasse un certain nombre de sous-éléments (qui lui appartiennent, mais ne sont pas imbriqués), on associe à chaque sous-élément un niveau $j \geqslant i + 1$ (si $i = 5$, on peut poser $j = 6$, ou $j = 7$, etc.).

Une chaîne PICTURE est destinée à décrire un texte qui doit se trouver dans la « case » correspondante d'une table.

Elle représente un mot formé de caractères dont chacun est affecté, dans la division des données, d'une signification déterminée. Les

numéros de niveaux et les chaînes PICTURE sont décrits en détail dans les points 10.6.1.1 et 10.6.1.2.

§ 10.3. Structure d'un programme COBOL

Un programme COBOL est composé des quatre *divisions* suivantes : la division de l'identification, la division de l'environnement, la division des données et la division de traitement. Le début d'une division est son en-tête, la fin, le dernier symbole précédant le début de la division suivante ou le dernier symbole du programme COBOL.

Une division se subdivise en *sections* ou en *paragraphes*. Le début d'une section est son en-tête, la fin, le dernier symbole de la division ou le dernier symbole précédant l'en-tête de la section suivante.

Chaque section, sauf les sections de la division des données, se compose de paragraphes. Le début d'un paragraphe est son nom, la fin, le dernier symbole précédant le paragraphe, la section, la division qui suit, ou le dernier symbole du programme COBOL.

La division de l'identification n'est pas découpée en sections, mais directement en paragraphes. Les divisions de l'environnement et des données sont découpées en sections. Les sections de la division de l'environnement sont subdivisées en paragraphes, celles de la division des données, en rubriques. En ce qui concerne la division de traitement, elle peut : ne pas être découpée en sections ou paragraphes, être découpée en paragraphes, être découpée en sections, se composer de quelques paragraphes suivis de quelques sections.

L'en-tête d'une division, d'une section, le nom d'un paragraphe se termine par un point.

Les noms des paragraphes faisant partie des divisions de l'identification, de l'environnement et des données sont fixes et représentent des mots réservés. C'est eux qui indiquent le début d'un paragraphe. Chaque paragraphe de l'une des divisions mentionnées contient, en plus de son nom, une ou plusieurs *phrases*. Chaque phrase se termine par un point. Une section de la division des données contient, en plus de son en-tête, une ou plusieurs phrases.

Chaque phrase représente une ou plusieurs clauses que l'on peut séparer par des virgules. Chaque clause dans une phrase est caractérisée par un mot réservé qui y figure et représente un élément sémantique et syntaxique de la phrase. La composition d'une clause sera décrite dans chaque cas concret.

Le nom d'un paragraphe de la division de traitement se compose d'un mot non réservé suivi d'un point. On reconnaît l'en-tête de paragraphe dans cette section par la présence d'un mot non réservé précédé d'un point et suivi d'une instruction.

Les paragraphes de la division de traitement sont eux aussi subdivisés en phrases (une ou plusieurs) dont chacune se termine par un point. Le début d'une phrase est le symbole suivant l'en-tête

de la division (s'il n'y a pas de sections ni paragraphes), d'un paragraphe, ou la fin de la phrase précédente.

Soulignons qu'une section comporte obligatoirement des paragraphes, ne soit-ce qu'un seul.

Une phrase de la division de traitement est une suite d'instructions (ou ordres). Chaque instruction commence par un verbe (qui est un mot réservé) et se termine par son dernier symbole qui précède une autre instruction ou le point qui termine la phrase.

§ 10.4. Division de l'identification

Cette division commence par son en-tête qui a la forme

|| IDENTIFICATION DIVISION. ||

et se compose de paragraphes dont chacun a son en-tête. L'en-tête d'un paragraphe est un mot réservé suivi d'un point. La division de l'identification peut contenir les paragraphes suivants:

|| PROGRAM-ID. || AUTHOR. || INSTALLATION. || DATE-WRITTEN. || DATE-COMPILED. || SECURITY. || REMARKS. ||.

Le premier paragraphe contient un nom de programme, le deuxième, un nom de programmeur; chaque paragraphe suivant peut contenir une ou plusieurs phrases dont la signification correspond au nom du paragraphe, le paragraphe || REMARKS || peut contenir n'importe quel texte. Le texte d'un paragraphe doit se terminer par un point. Tous les paragraphes sont optionnels sauf le premier qui est obligatoire. Il vient toujours le premier, les autres peuvent être spécifiés dans n'importe quel ordre.

Exemple 10.12. Voici la division de l'identification d'un programme COBOL:

```
|| IDENTIFICATION DIVISION.
 PROGRAM-ID. CALCUL CENTRALISE.
 AUTHOR. IVANOV-A-P.
 DATE-WRITTEN. 2 MAI 1974.
 REMARKS. LE PROGRAMME EST DESTINE AU
 CALCUL CENTRALISE DES BESOINS EN FER LAMINE. ||.
```

Le paragraphe DATE-COMPILED, s'il est spécifié, fournit la date du jour sur le listing au moment de la compilation, indépendamment de ce qu'il y était écrit avant.

§ 10.5. Division de l'environnement

Cette division commence par son en-tête

||ENVIRONMENT DIVISION.||

et se compose de deux sections qui commencent par leurs en-têtes, formés des mots réservés respectifs

|| CONFIGURATION SECTION.|| INPUT — OUTPUT SECTION ||.

Chaque section est divisée en paragraphes dont les noms sont fixes.

10.5.1. Section de la configuration. Cette section comprend les paragraphes nommés par les mots réservés suivants: || SOURCE-COMPUTER.|| OBJECT-COMPUTER.|| SPECIAL-NAMES.|| . Les deux premiers paragraphes précisent sur quels calculateurs le programme sera compilé, assemblé et exécuté.

Le paragraphe SPECIAL—NAMES relie les noms de certains dispositifs de la machine d'exécution à des noms mnémoniques utilisés dans le programme—source. Il se compose d'une ou de plusieurs phrases, formée chacune par un nom de dispositif suivi du mot réservé || IS|| suivi d'un nom mnémonique non réservé par lequel l'utilisaé teur veut nommer ce dispositif dans le programme COBOL.

Exemple 10.13. Voici la section de la configuration d'un proé ramme COBOL:

|| ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-360-H50.
OBJECT-COMPUTER. IBM-360-H50 MEMORY
SIZE 50 000 CHARACTERS.
SPECIAL-NAMES. SYSOUT IS IMPRY. C01 IS SAUT.||.

Pour imprimer une donnée TOTAL par exemple, on peut se servir, dans la division de traitement, de l'instruction

|| DISPLAY TOTAL UPON IMPRY||.

Pour fournir la même donnée sur le pupitre de l'ordinateur, on peut écrire l'instruction

|| DISPLAY TOTAL UPON CONSOLE||,

|| CONSOLE|| étant le mot réservé désignant le pupitre.

Toute la section de la configuration, ainsi que chaque son paragraphe, sont optionnels. Si elle n'est pas spécifiée, les SOURCE-COMPUTER et OBJECT-COMPUTER sont une même machine.

Pour plus de détail sur la structure de la CONFIGURATION SECTION il faut consulter les mannuels d'utilisation de calculateurs particuliers.

10.5.2. Section d'entrée-sortie. Cette section se compose des paragraphes nommés par les mots réservés suivants:

|| FILE-CONTROL.|| I-O-CONTROL.||.

Ces notations signifient respectivement: contrôle de fichiers et contrôle d'entrée-sortie.

Le paragraphe FILE-CONTROL contient autant de phrases qu'il y a de fichiers à traiter (une phrase par fichier). Chaque phrase commence par une instruction SELECT qui peut être suivie des clauses FOR MULTIPLE, RESERVE, ACCES, PROCESSING MODE.

Une instruction SELECT contient le nom symbolique d'un fichier et sert à décrire les supports d'information et les unités d'entrée-sortie assignés au fichier.

Une instruction SELECT se compose d'un mot réservé || SELECT|| suivi d'un nom de fichier (un nom externe) suivi d'un mot réservé || ASSIGN TO|| suivi d'une liste de noms externes spécifiant les dispositifs assignés au fichier.

La clause FOR MULTIPLE peut être spécifiée (facultativement) lorsque le fichier en question est enregistré sur plusieurs bandes ou sur plusieurs disques. Elle aura respectivement la forme || FOR MULTIPLE REEL|| ou || FOR MULTIPLE UNIT||.

La clause RESERVE peut être spécifiée pour un fichier dont l'organisation est indexée et dont le mode d'accès est séquentiel. Cette clause représente une suite formée par le mot || RESERVE||, un nombre entier ou un mot réservé || NO|| et les mots || ALTERNATE AREA|| (ou ||ALTERNATE AREAS||). Elle indique que, pour le fichier donné, un certain nombre de tampons (zones d'entrée-sortie) représenté par un nombre entier doit être réservé en plus du tampon qu'affecte automatiquement le système.

Nous nous passons de la description d'autres clauses pouvant figurer dans la phrase commencée par une instruction SELECT.

Exemple 10.14. Un paragraphe FILE-CONTROL peut avoir la forme:

```
|| FILE—CONTROL.
   SELECT DONNEES-INITIALES ASSIGN TO UT-2314-S-FCART
  RESERVE 3 ALTERNATE AREAS
  ACCES IS SEQUENTIAL.
  SELECT TOTAL ASSIGN TO UR-1403-S-EDI.
  SELECT ARTICLES ASSIGN TO UR-1405-IMPRIMTE.||.
```

Le paragraphe I-O-CONTROL sert à: faire enregistrer, aux moments désirés de l'exécution du programme, l'état de la mémoire centrale, afin de pouvoir reprendre le déroulement du programme à partir d'un tel moment en cas d'une erreur-machine; faire partager entre plusieurs fichiers une même zone de travail en mémoire; indiquer les fichiers occupant une même bande. Ces objectifs sont réalisés par les clauses respectives RERUN, SAME AREA, MULTIPLE FILE TAPE. Chacune d'elles représente une phrase.

La clause RERUN représente les mots || RERUN ON || suivis d'un nom-de-fichier-1 puis du mot || EVERY ||, puis d'un entier non signé, puis des mots || RECORDS OF || et enfin d'un nom-de-fichier-2.

La clause SAME AREA commence par les mots || SAME RECORD AREA FOR ||, ou || SAME SORT AREA FOR || ou || SAME AREA FOR ||. Ces mots sont suivis d'une liste de noms-de-fichiers. Les options énumérées précisent le type de la zone que les fichiers donnés se partagent en mémoire.

La clause MULTIPLE FILE TAPE permet de consulter ou de créer des fichiers d'organisation séquentielle standard sur une bande magnétique multi-fichiers. Le format général comprend les mots || MULTIPLE FILE TAPE CONTAINS || suivis d'une liste de noms-de-fichier.

Tout le paragraphe I-O-CONTROL peut être omis si le programmeur n'en a pas besoin.

§ 10.6. Division des données

La division des données commence par son en-tête

|| DATA DIVISION. ||

et se subdivise en quatre sections: la section des fichiers (FILE SECTION), la section des mémoires de travail (WORKING-STORAGE SECTION), la section de liaison (LINKAGE SECTION), la section d'édition (REPORT SECTION).

Chaque section peut comporter un nombre de descriptions dont chacune représente une phrase. Une section sans description peut être omise.

10.6.1. Notions utilisées pour la description des données. Les données traitées par un programme COBOL doivent être représentées sous forme d'articles (enregistrements) qui sont réunis en fichiers. On distingue deux types de subdivisions d'un article: la donnée élémentaire qui n'est plus subdivisée et qui doit être décrite dans une clause PICTURE, et le groupe qui est lui-même subdivisé en groupes, ou en groupes et données élémentaires, ou en données

élémentaires. Un groupe est considéré comme décrit dès qu'on a la description de toutes les données qu'il réunit et sa propre description.

10.6.1.1. Notion de niveau. L'hiérarchie des données qui composent un article est interprétée par les numéros de niveaux (p. 10.2.2). On convient d'associer à l'article lui-même le numéro de niveau 01. S'il y a une donnée dont le numéro de niveau est i, alors toutes les données qu'elle réunit ont un numéro de niveau j satisfaisant à la condition $j > i$. En particulier, on peut avoir $j = i + 1$.

Une fois les numéros des niveaux attribués, la description des données d'un article se fait comme suit :

1) On considère l'article comme une donnée; on le décrit.

2) On vérifie si la donnée en considération contient d'autres données qui ne sont pas encore décrites. Si oui on passe à la première d'entre elles; on la décrit et l'on reprend 2. Si non on passe à 3.

3) Si le numéro de niveau de la donnée en considération est 01, on termine le procédé; si non on passe à 4.

4) On abandonne la donnée en considération en passant à la donnée qui la contient. On reprend 2.

Exemple 10.15. Un article d'une structure schématisée par la table 10.1, ainsi que toutes les données qu'il contient, seront décrits dans l'ordre suivant :

01a, 02b, 04e, 04f, 02c, 03g, 03h, 04i, 04j, 04k, 02d.

198 *Table 10.1*

a						
b		c				d
			h			
e	f	g	i	j	k	

Les numéros de niveau des données d'un enregistrement prennent les valeurs de 01 à 49.

De plus, il existe deux niveaux spéciaux, avec les numéros de niveaux 77 et 88.

Le niveau 77 est associé, à la place de 01, à une donnée indépendante qui n'est pas subdivisée et n'appartient à aucun groupe. Une telle donnée ne peut pas apparaître dans la section des fichiers (voir les pp. 10.6.4 et 10.6.5). Le niveau 88 sert à la description de noms

de condition (noms associés aux valeurs particulières d'une variable conditionnelle, p. 10.8.2.).

10.6.1.2. Chaîne PICTURE. On a déjà dit que chaque donnée élémentaire doit être décrite par une chaîne PICTURE. On distingue plusieurs types (schémas) des chaînes PICTURE, permettant de spécifier la classe, la longueur et les éventuelles particularités des données élémentaires. Les schémas d'édition décrivent les données qui sont à imprimer et ne s'utilisent pas comme données initiales ou intermédiaires. Les autres schémas (alphabétique, alphanumérique et numérique) décrivent les données intiales et intermédiaires.

Un schéma de description d'une donnée alphabétique, alphanumérique ou numérique représente une suite de symboles qui peut contenir: a) les symboles || 9|| A || X || S|| P|| qui indiquent que la position occupée par le symbole est (ou sera) occupée, dans la donnée élémentaire en question, respectivement par: un chiffre, une lettre (ou un espace), n'importe quel caractère du code EBCDIC, un signe algébrique, un zéro; b) le symbole || V|| qui indique la position du point décimal virtuel dans la suite des chiffres qui compose le nombre défini. Bien que ce symbole occupe une place dans le schéma, il n'entraîne pas la réservation d'espace dans le texte qui représente la valeur de la donnée.

Lorsqu'un symbole figure dans une chaîne-de-caractères plusieurs fois de suite, on peut alléger l'écriture en l'écrivant une seule fois et en le faisant suivre du nombre d'exemplaires de ce symbole entre parenthèses. Par exemple, on écrit A(8) à la place de AAAAAAAA.

Un schéma alphabétique définit une donnée élémentaire se composant de lettres et d'espaces; il représente une suite de symboles A.

Un schéma alphanumérique définit une donnée élémentaire alphanumérique. Il comprend une suite des symboles A, X et 9. La suite est traitée comme si elle ne contenait que des X.

Pour définir une donnée numérique point fixe il faut que sa chaîne contient: un symbole 9 à chaque position correspondant à un chiffre; un symbole S à la position du signe (si un signe doit être présent dans la donnée correspondante); un symbole V entre les groupes de symboles 9, là où un point décimal doit se trouver; un symbole P à chaque position où un zéro est sous-entendu (au début ou à la fin de la chaîne). Dans ce cas un point décimal est supposé placé avant le premier P (si les P sont à gauche dans la chaîne) ou après le dernier P (s'ils sont à droite dans la chaîne).

Bien que les symboles P occupent de la place dans le schéma, ils ne demandent pas de réservation de mémoire dans le champ destiné à la valeur de la donnée.

Pour définir une donnée numérique point flottant on écrit une chaîne qui se compose de symboles || S|| désignant les signes de la

mantisse et de l'exposant, de symboles || 9|| et d'un symbole || E|| séparant la mantisse et l'exposant. Deux positions sont réservées à l'exposant.

EXEMPLE 10.16. La table présente quelques exemples qui montrent comment un schéma interprète la valeur d'une donnée élémentaire.

Table 10.2

Chaînes PICTURE de données alphabétiques, alphanumériques et numériques

Donnée élémentaire stockée en mémoire	Chaîne PICTURE interprétant la donnée	Valeur de la donnée
DATE	A(4)	DATE
B314	X(4)	B314
92835	9(2)V9(3)	92.835
8312	9V999	8.312
52731	P(2)9(5)	.0052731
52731	9(5)P(2)	5273110.
−52731	S9(2)V9(3)	−52.731
52731	S9(2)V9(3)	52.731
−527	SP(2)9(3)	−.00527
+25E−3	S9(4)S99	+2500E−03
+00025E−3	S9(4)S99	+0025E−04

Dans le cas des données éditées, un schéma d'édition peut contenir, en plus des symboles déjà vus, les symboles d'édition suivants :

a) les symboles d'insertion || 0|| .|| ,|| B|| $ || qui désignent respectivement l'insertion d'un zéro, d'un point décimal, d'une virgule, d'un espace, du symbole monétaire; de plus, peut être inserée n'importe quelle notation non numérique ;

b) les symboles de substitution || Z|| * || —|| +|| qui désignent respectivement la suppression d'un zéro qui occupe la même position que Z (ou *) s'il se trouve à gauche de la partie entière d'un nombre et le remplacement par un espace ou un astérisque ; l'impression d'un « moins » si le nombre est négatif, et d'un espace s'il est positif ; l'impression d'un « moins» si le nombre est négatif, et d'un « plus » s'il est positif.

EXEMPLE 10.16a. La table 10.3 illustre la signification des symboles d'insertion.

EXEMPLE 10.16b. La table 10.4 illustre l'utilisation des symboles de substitution

10.6.2. Description de données.

10.6.2.1. Structure d'une description. Une description de donnée représente une phrase et peut comprendre les clauses suivantes (pouvant être séparées par des virgules) :

1) l'identification de la donnée (est obligatoire pour toute description de donnée).
2) La clause REDEFINES.
3) La clause PICTURE (est absente dans la description d'un groupe, obligatoire dans la description d'une donnée élémentaire).
4) La clause VALUE.
5) La clause OCCURS.
6) La clause USAGE.
7) La clause BLANK.

L'ordre de succession des clauses, à partir de la troisième, peut être modifié.

Table 10.3

Chaînes PICTURE d'édition de données numériques par la méthode d'insertion

Valeur d'une donnée	Chaîne PICTURE	Edition
+913	−999	⊔913
−913	−999	−913
913	−999	⊔913
0221 (point décimal implicite après 0)	+9.99	+0.22
6281	9(3)BB99	062⊔⊔81
2565	99.99	$ 25.65
−5	−9 'kg'	−05 kg

L'identification d'une donnée commence par un numéro de niveau suivi soit par un nom de donnée, soit par le mot ‖ FILLER ‖. Celui-ci s'utilise pour la description d'une donnée élémentaire qui n'est pas mentionnée dans la division de traitement (puisque ne s'emploie que dans un groupe de données), mais exige un emplacement dans la mémoire. Une telle donnée ne nécessite donc pas un nom. Par conséquent, toute description contenant le mot FILLER contient forcément une clause PICTURE.

La clause REDEFINES est inclue dans une description pour fournir au compilateur l'information qui lui permettrait de gérer

Table 10.4

Chaînes PICTURE d'édition de données numériques par la méthode de suppression et remplacement

Valeur d'une donnée	Chaîne PICTURE	Edition
00205	ZZZ99	⊔⊔ 205
00041	Z(2)9(3)	⊔⊔ 041
3681	Z(2)99	3681
0395	* 9(3)	* 395
0048	* 999	* 048
2985	** 99	2985
0298	** 99	* 298
0029	** 99	** 29
0002	** 99	** 02
00098	———99	⊔⊔⊔ 08
00753	++999	⊔ +753
—00753	++999	⊔ —753
00000	ZZ999	⊔⊔ 000

d'une manière efficace la mémoire en affectant une même zone de mémoire aux données différentes (qui ne peuvent, bien entendu, s'y trouver simultanément). La clause REDEFINES commence par le numéro-de-niveau commun à deux noms-de-donnée : un nom-de-donnée-1 qui est le nom utilisé dans la clause de l'identification, un nom-de-donnée-2 qui désigne la même zone de mémoire que le nom-de-donnée-1, mais ne lui prête pas les mêmes caractéristiques. Le numéro-de-niveau commun est suivi de nom-de-donnée-1, puis du mot || REDEFINES || et ensuite du nom-de-donnée-2.

En utilisant dans la description d'une donnée la clause REDEFINES les règles suivantes sont à estimer :

a) les clauses REDEFINES redéfinissant une zone de mémoire doivent suivre immédiatement la description initiale de cette zone ;

b) la description de nom-de-donnée-1 ne peut contenir la clause OCCURS ;

c) les clauses redéfinissant une zone de mémoire (qui concernent nom-de-donnée-2 ou une donnée subordonnée à nom-de-donnée-2) ne peuvent contenir de clause VALUE ;

d) la redéfinition commence à nom-donnée-2 et finit lorsqu'on rencontre un numéro-de-niveau numériquement inférieur ou égal à celui de nom-de-donnée-2.

Exemple 10.17. La section des mémoires de travail de la division des données (voir le p. 10.6.4) peut contenir le fragment suivant :

||01 COUPLE.
02 ABSCISSE PICTURE 99.
02 ORDONNEE PICTURE 99.
02 COTE PICTURE 999.
01 VECTEUR-BIDIMENSIONNEL REDEFINES COUPLE.
02 COMPOSANTE-X PICTURE 999.
02 COMPOSANTE-Y PICTURE 999.
01 SCALAIRE PICTURE 9999.||.

Soulignons que les niveaux et les longueurs satisfont ici aux conditions imposées plus haut : les numéro-de-niveau sont égaux, la longueur de nom-de-donnée-1 VECTEUR-BIDIMENSIONNEL (mesurée en octets qu'il faut pour stocker la chaîne PICTURE 999999) ne dépasse pas celle de nom-de-donnée-2 COUPLE (qui correspond à la chaîne PICTURE 9(7)). Il serait incorrect de mettre la phrase

||01 SCALAIRE PICTURE 9999.||

avant la phrase

||01 VECTEUR-BIDIMENSIONNEL REDEFINES COUPLE.||

car la description de nom-de-donnée-2 doit suivre immédiatement celle de nom-de-donnée-1.

La suite de phrases donnée au début de cet exemple ne peut faire partie de la section des fichiers de la division des données (voir le p. 10.6.3), puisque dans cette section les données redéfinie et redéfinissante doivent non seulement avoir une même longueur (ce qui n'a pas lieu ici), mais aussi une même structure. En outre, l'utilisation de la clause REDEFINES au niveau 01 dans la section des fichiers est à éviter.

La clause PICTURE représente une notation formée par le mot ||PICTURE|| suivi d'une chaîne PICTURE (voir le p. 10.6.1.2 et l'exemple 10.17).

La clause VALUE sert à donner une valeur initiale à une donnée qui, pour une raison ou une autre, n'est pas introduite dans le calculateur ni formée au moyen d'instructions de la PROCEDURE DIVISION. Cette clause n'est utilisée que dans : a) la section des mémoires de travail ; b) la section des fichiers, pour une donnée faisant partie d'un fichier à imprimer.

La clause considérée se compose des mots ||VALUE IS|| suivis d'un littéral.

Si la clause VALUE s'emploie avec la clause PICTURE, il ne doit pas y avoir de contradiction. Ainsi, lorsqu'une donnée spécifiée dans la chaîne PICTURE est numérique, il doit en être de même du littéral dans la clause VALUE. La clause VALUE ne s'utilise pas au niveau de groupe de données.

EXEMPLE 10.18. Donnons trois exemples d'utilisation de la clause VALUE :

```
|| 02 FOURNISSEUR PICTURE A (10) VALUE
   IS 'USINE ␣ VEF ␣'.||.
|| 04 SOMME PICTURE 9V99 VALUE IS 001.||.
|| 02 FILLER PICTURE A (6) VALUE IS ALL '␣'.||.
```

Dans le dernier exemple le littéral est représenté par une constante figurative.

La clause OCCURS sert à éviter les répétitions en décrivant des données de nature et de niveau identiques qui sont organisées en tables, vecteurs ou matrices.

La clause est formée par le mot || OCCURS || suivi d'un entier non signé suivi du mot || TIMES ||; cette construction peut être elle-même suivie des mots || INDEXED BY || précédant un nom-index. Celui-ci ne demande aucune description puisqu'il n'est pas une donnée.

Cette clause ne s'utilise pas au niveau 01. L'entier ne dépasse pas 1023.

Si la clause OCCURS s'utilise pour la description d'un groupe de données, le nombre de descriptions des données élémentaires formant le groupe qui se répète ne doit pas dépasser 63 (32 pour un fichier d'organisatien indexée et dont le mode d'accès est séquentiel).

La clause OCCURS peut s'employer à différents niveaux de description d'un article. Le nombre de ces niveaux ne doit pas dépasser 3.

EXEMPLE 10.19. La description

```
|| 02 ETABLISSEMENT OCCURS 10 TIMES.
   03 DEPARTEMENT OCCURS 5 TIMES.
   04 GRADE OCCURS 3 TIMES.
   05 NOM OCCURS 20 TIMES.|| . . . est inadmissible,
```

puisque le nombre de niveaux dépasse 3 (est égal à 4).

EXEMPLE 10.20. Une liste de la forme

Numéro	Nom	Salaire

contenant vingt lignes peut être décrite comme suit :

```
|| 01 LISTE.
   02 LIGNE OCCURS 20 TIMES.
   03 NUMERO PICTURE 99.
   03 NOM PICTURE A (10).
   03 SALAIRE PICTURE 999V99.||.
```

EXEMPLE 10.21. Une table de la forme

Ecole	Nombre d'élèves		
	1973	1974	1975

contenant 15 lignes peut être décrite comme suit :

```
|| 01 TABLE.
   02 LIGNE OCCURS 15 TIMES INDEXED BY I.
   03 ECOLE PICTURE A (20).
   03 NOMBRE-D-ELEVES.
   04 ANNEE PICTURE 9999 OCCURS 3 TIMES
   INDEXED BY J.||.
```

Le nom-de-donnée qui représente un ensemble de données décrit par OCCURS doit être indicé chaque fois qu'il est utilisé dans la PROCEDURE DIVISION. L'indice est un nombre entier ou le nom d'une donnée entière écrit entre parenthèses, à la suite de nom-de-donnée. Les indices multiples sont toujours écrits de gauche à droite, dans l'ordre décroissant de l'appartenance. Ils sont mis dans une

seule paire de parenthèses et séparés par un espace ou par une virgule et un espace.

La clause USAGE commence par les mots || USAGE IS||. Ils sont suivis de l'une des notations : || DISPLAY COMPUTATIONAL|| || COMPUTATIONAL-1|| COMPUTATIONAL-2|| COMPUTATIONAL-3|| INDEX||.

La clause USAGE précise le format d'une donnée en mémoire. Si elle n'est pas spécifiée, l'option DISPLAY est prise par omission. La clause s'utilise pour une donnée de n'importe quel niveau. Au niveau du groupe, elle s'applique à toutes les données élémentaires appartenant au groupe.

L'option COMPUTATIONAL décrit un littéral point fixe (représenté dans la machine en code binaire). L'option COMPUTATIONAL-n (où $n = 1, 2$) décrit un littéral numérique représenté en point flottant (en simple précision pour $n = 1$ et en double précision pour $n = 2$). L'option COMPUTATIONAL-3 définit une donnée décimale représentée sous forme condensée, c'est-à-dire avec un demi-octet par position décimale, plus un demi-octet pour le signe. L'option INDEX peut figurer à n'importe quel niveau. Elle indique que la donnée spécifiée servira d'index lors de la recherche d'un élément dans une table.

La clause BLANK a la forme

|| BLANK WHEN ZERO||.

Elle indique que, si la donnée numérique décrite a une valeur nulle, les espaces remplacent les zéros lors de l'impression. La clause n'est donc utilisable qu'au niveau élémentaire.

10.6.2.2. Déclaration d'un nom-de-condition au niveau 88. Les descriptions de données s'utilisent dans les sections des fichiers, des mémoires de travail et de liaison de la division des données.

Dans la section des fichiers, les descriptions de données apparaissent comme des parties composantes de descriptions d'articles. Dans la phrase définissant la première donnée de la description d'un article figure toujours le numéro de niveau 01. Elle est suivie, dans l'ordre indiqué au p. 10.6.1.1, de phrases de niveaux supérieurs.

Notons que la description d'une donnée élémentaire (qui contient une clause PICTURE) peut être immédiatement suivie d'une ou de plusieurs phrases spéciales qui servent à définir des variables logiques (noms-de-condition) associées à la donnée élémentaire en question. Le numéro de niveau dans une telle phrase est 88. Il est suivi d'un nom-de-condition, puis des mots || VALUE IS||, et enfin d'un ou de plusieurs littéraux séparés par des virgules.

La variable logique ainsi définie prend la valeur « vrai » si la donnée élémentaire correspondante a pour valeur l'un des littéraux qui figurent dans la phrase.

Exemple 10.22. L'utilisation d'un nom-de-condition peut être illustrée par l'exemple suivant :

```
|| 01 FEUILLE-DE-PAYE.
02 ATELIER PICTURE A (9).
88 ATELIER-CHAUD VALUE IS '␣␣␣␣FORGE',
ESTAMPAGE'.
02 MOIS PICTURE 99.
88 JANVIER VALUE IS 01.
88 FEVRIER VALUE IS 02.
02 LIGNE OCCURS 40 TIMES.
03 NOM-PRENOM PICTURE A (30).
03 NET PICTURE 9 (5).||.
```

Dans cette description, la variable logique ATELIER-CHAUD est donnée par une phrase de niveau 88. Cette variable prend la valeur « vrai » si la feuille de paye contient le nom d'atelier FORGE ou ESTAMPAGE. En cas d'un autre nom d'atelier, la variable logique prend la valeur « faux ».

De plus, dans notre exemple, sont définies à l'aide de phrases de niveau 88 les variables logiques JANVIER et FEVRIER. Elles deviennent vraies lorsque la donnée MOIS prend respectivement la valeur 01 ou 02.

10.6.2.3. Phrases de niveau 77. La section des mémoires de travail de la division des données peut contenir des descriptions des données servant à définir les zones de travail utilisées par le programme. Une zone, lorsqu'elle est décrite comme un article, a pour description plusieurs phrases dont la première est de niveau 01 et les dernières contiennent les PICTURE. Si une zone est décrite comme une donnée indépendante, sa description est une phrase commencée par le numéro-de-niveau 77. Une phrase de niveau 77 ne doit pas contenir de clauses REDEFINES, OCCURS, BLANK. Au contraire, la clause d'identification et celle PICTURE y figurent obligatoirement.

Exemple 10.23. La description d'une donnée indépendante peut avoir l'une des formes suivantes:

```
|| 77 QUANTITE-1 PICTURE 9 (20) COMPUTATIONAL.||
77 QUANTITE-2 PICTURE X(10) VALUE IS 'USINE␣ZIL␣'.||.
```

Une phrase de niveau 77 peut être utilisée pour décrire une zone réservée à stocker un index. Dans ce cas, en plus de la clause identifiant la donnée (l'index), la phrase doit contenir la clause USAGE IS INDEX. La clause PICTURE peut être omise.

EXEMPLE 10.24. Voici deux exemples de description d'une zone réservée à l'index :

|| 77 NUMERO INDEX. || 77 IND-1 PICTURE 99 USAGE IS INDEX VALUE IS 25. ||.

La seconde phrase stipule que la valeur initiale d'un index de nom IND-1 est 25.

10.6.2.4. Nom qualifié. Il arrive que certains groupes de données contiennent, aux niveaux supérieurs, des données de même nom. De telles données sont impossibles à utiliser par les instructions du programme COBOL. On peut alors préciser un nom-de-donnée ambigu, le qualifier comme on dit. A cette fin, on le fait suivre du mot || IN || (ou || OF ||), puis du nom du groupe auquel (qu'on appelle qualificateur) appartient directement la donnée en question. Ceci fait, si le nom qualifié reste toujours ambigu, on peut ajouter encore un || IN || (ou || OF ||) suivi du nom du groupe auquel appartient directement la donnée précédant le dernier || IN ||, etc.

Les données de niveau 01 doivent obligatoirement avoir un nom unique.

EXEMPLE 10.25. Supposons que la donnée qu'il faut décrire ait la structure suivante :

Feuille de paie

	1972		1973	
	Nom	Total	Nom	Total
1				
2				

On peut décrire cette donnée de la manière suivante :

```
|| 01 TABLE.
02 LIGNE OCCURS 40 TIMES.
03 72-ANNEE.
04 NOM PICTURE A(30).
04 TOTAL PICTURE 9(5).
03 73-ANNEE.
04 NOM PICTURE A (30).
04 TOTAL PICTURE 9 (5). ||.
```

S'il faut appeler la donnée élémentaire TOTAL qui fait partie du groupe 72-ANNEE, on peut effectuer une qualification qui fournira le nom TOTAL IN 72-ANNEE.

10.6.3. Section des fichiers. Cette section a pour l'en-tête la phrase

|| FILE SECTION. ||

et se compose de descriptions de fichiers suivies chacune de descriptions des enregistrements (articles) du fichier.

10.6.3.1. Description de fichier. Une description de fichier constitue une phrase qui comprend les clauses suivantes :
1) La clause de l'identification du fichier (obligatoire).
2) La clause RECORDING MODE (option).
3) La clause BLOCK CONTAINS (option).
4) La clause LABEL RECORD (obligatoire).

La clause de l'identification représente l'indicateur de niveau || FD || (File Description) suivi d'un nom-de-fichier.

La clause RECORDING MODE spécifie le format des articles contenus dans le fichier objet de la clause. On écrit les mots || RECORDING MODE IS|| suivis de l'une des lettres || F|| , || V|| ou || U||. Un F (fixe) signifie que les enregistrements du fichier donné sont tous de même longueur fixe. Les deux autres lettres spécifient deux types de formats d'enregistrements de longueur variable. En règle générale, la clause RECORDING MODE doit être spécifiée si les articles du fichier considéré sont de type F ou U ; si elle est absente, les articles seront supposés être de type V.

La clause optionnelle BLOCK CONTAINS caractérise la façon de subdiviser le fichier donné en parties appelées blocks. Un block contient les informations disponibles en mémoire dans une zone tampon qui sert de zone de lecture ou d'écriture pour le système d'exploitation. La clause représente les mots || BLOCK CONTAINS|| suivis d'un entier suivi du mot || CHARACTERS|| (ou ||RECORDS||). L'entier précise le nombre d'octets occupés en mémoire par les caractères d'un block, si l'on choisit l'option CHARACTERS; en cas de RECORDS, l'entier exprime le nombre d'enregistrements de longueur maximale que peut contenir un block. Lorsque la clause BLOCK CONTAINS est omise, le fichier est considéré comme n'étant pas bloqué.

La clause LABEL RECORD, dont la présence est obligatoire dans toute rubrique de description de fichier, est utilisée pour définir le mode d'étiquetage du fichier considéré. Elle indique l'absence ou la présence d'étiquette sur un fichier, et le type d'étiquette utilisée. La clause représente les mots || LABEL RECORD IS|| ou || LABEL RECORDS ARE|| suivis ou bien du mot || STANDARD||, ou bien du mot || OMITTED||, ou bien d'un nom-de-donnée. L'option STAN-

DARD signifie que le fichier considéré comporte un système complet d'étiquettes conformes au standard adopté pour la machine donnée. Cette option exclut le traitement des étiquettes créées par l'utilisateur. L'option OMITTED est obligatoire pour les fichiers assignés à des équipements à enregistrement unique déclarés UR dans la division de l'environnement. L'option « nom-de-donnée » implique le traitement, en plus d'étiquettes standard, d'une étiquette créée par l'utilisateur.

10.6.3.2. Descriptions d'enregistrements. On a déjà dit que chaque description de fichier est suivie de descriptions des enregistrements de ce fichier. Chaque description d'enregistrement commence par une phrase de niveau 01.

La clause VALUE est interdite dans la section des fichiers dans son utilisation normale pour préciser une valeur initiale. Elle ne peut apparaître qu'associée à un nom-de-condition (au niveau 88). La clause REDEFINES ne s'utilise pas dans la section des fichiers au niveau 01.

10.6.4. Section des mémoires de travail. Cette section est destinée à décrire les zones de travail où sont stockées, au cours de l'exécution du programme, les données intermédiaires. Les zones de travail sont décrites dans les rubriques de description des données qui y seront mises.

La section des mémoires de travail a pour en-tête la phrase

|| WORKING-STORAGE SECTION. ||.

Cette phrase est suivie d'abord de descriptions de données indépendantes (commençant par le numéro-de-niveau 77), puis de descriptions d'enregistrements (commençant par le numéro-de-niveau 01). Il est à signaler que

1) il est permis d'utiliser la clause REDEFINES au niveau 01;

2) il n'est pas exigé que la longueur d'une donnée redéfinissante soit exactement égale à celle de la donnée à redéfinir; il suffit que la première longueur ne dépasse pas la deuxième;

3) on recourt obligatoirement à la clause VALUE pour les noms-de-condition et on peut l'utiliser pour attribuer une valeur initiale aux éléments d'une mémoire de travail;

4) la clause VALUE peut être utilisée au niveau de groupe; alors son littéral doit être une constante figurative ou un littéral non numérique, la zone réservée au groupe en question sera remplie sans tenir compte de sa division en zones correspondant aux données élémentaires;

5) la clause OCCURS ne peut être utilsée aux niveaux 01 et 77.

10.6.5. Section de liaison. Cette section décrit les données utilisables par le programme bien que définies à l'extérieur de celui-ci. Outre la section de liaison, ces données doivent être décrites soit dans la section des fichiers, soit dans celle des mémoires de travail (puisque les informations contenues dans la section de liaison ne sont pas utilisées par le compilateur au cours de l'allocation de mémoire).

La section de liaison a pour en-tête la phrase

||LINKAGE SECTION.||

qui est suivie de descriptions des données indépendantes (de niveau 77), puis de descriptions d'enregistrements. La composition de descriptions est soumise aux mêmes règles que pour la section des mémoires de travail.

§ 10.7. Langage des opérandes lié au COBOL

Les données initiales, de même que les résultats intermédiaires et définitifs d'un programme COBOL, constituent des *états de mémoire* dont chacun réunit un *état de mémoire extérieure* et un *état de mémoire intérieure.*

Les auteurs du COBOL ne donnent aucune description du langage des opérandes lié au COBOL. On peut concevoir un tel langage des opérandes comme suit.

L'alphabet du langage des opérandes comprend tous les symboles de l'alphabet du COBOL plus trois symboles complémentaires que nous choisirons de la forme ||≡||∘||∇||.

Un état de mémoire extérieure représente un ensemble des états de fichiers. On peut écrire chaque état de fichier sous forme d'une chaîne formée par un nom de fichier, un symbole ||≡|| et une suite de notations, chaque notation se terminant par le symbole ||∇||. Une notation représente une suite de mots machine séparés par des symboles ||∘||. Chaque mot machine représente une valeur d'une donnée dont la description contient une clause PICTURE et ne contient pas de clause VALUE (la clause VALUE si elle est contenue dans la description d'un groupe auquel la donnée considérée appartient est sans effet sur cette donnée). Si la description de la donnée utilise une clause OCCURS n TIMES, alors la chaîne contient n mots d'utilisateur en cas d'une donnée élémentaire, ou n groupes de mots d'utilisateur en cas où la donnée est un groupe.

EXEMPLE 10.26. Si la section des fichiers de la division des données contient la description de fichier

||FD FICHPAIE LABEL RECORD IS STANDARD.
01 FEUILLE-DE-PAYE.
02 ATELIER PICTURE A (9).

88 ATELIER-CHAUD VALUE IS '⊔⊔⊔⊔FORGE', 'ESTAMPAGE'.
02 MOIS PICTURE 99.
88 JANVIER VALUE IS 01.
02 LIGNE OCCURS 5 TIMES.
03 NOM-PRENOM PICTURE A (14).
03 NET PICTURE 999V99.
03 COMMENTAIRE PICTURE A (10)
VALUE IS ALL '⊔'.||,

alors on peut écrire pour elle un état de fichier

|| FICHPAIE ≡ ⊔⊔MONTAGE$_\circ$01$_\circ$⊔⊔⊔EVANS-JULES$_\circ$130.12$_\circ$FOURIER-CLAUDE$_\circ$100.00$_\circ$⊔ LANSON-NESTOR$_\circ$ 224.01$_\circ$⊔⊔⊔HOUDON-JEAN$_\circ$ 114.10$_\circ$⊔MURAT-JOACHIM$_\circ$098.50∇ESTAMPAGE$_\circ$01$_\circ$⊔ CYRIL-LE-ANDRE$_\circ$075.25$_\circ$DANDREAS-CHRIS$_\circ$112.74$_\circ$DELACOURS-ALIC$_\circ$099.12$_\circ$IGNATIEV-JULES$_\circ$118.02$_\circ$⊔⊔⊔IGUAZU-PAUL$_\circ$ 117.18∇||.

Cet état de fichier contient dans sa partie droite (à droite du symbole || ≡ ||) deux notations dont chacune exiprime une feuille de paie.

Un état de mémoire intérieure représente un ensemble d'*états de données élémentaires* (nous appliquerons ce terme aussi bien aux résultats intermédiaires et définitifs élémentaires). Tout état de donnée élémentaire est une chaîne formée par un nom de donnée élémentaire, un symbole || ≡ ||, un mot d'utilisateur qui représente une valeur de la donnée élémentaire, et un symbole || $_\circ$|| qui termine la chaîne. Le nom de donnée élémentaire ou bien coïncide avec un nom-de-donnée dont la description dans la section des fichiers ou la section des mémoires de travail de la division des données contient une clause PICTURE ; ou bien, si la description mentionnée contient aussi les mots INDEXED BY (ou appartient à un groupe décrit avec INDEXED BY), contient en plus une liste de valeurs d'indices entre parenthèses (cette liste comportant un, deux ou trois entiers non signés, séparés par des virgules). Si le nom-de-donnée demande une qualification, alors, dans la description d'état de la donnée élémentaire, il doit déjà être qualifié. Rappelons que, dans certains cas, le rôle du nom de donnée élémentaire est joué par le mot réservé FILLER (voir le p. 10.6.2).

Exemple 10.27. Reprenons l'exemple 10.26 avec cette différence que la description du groupe LIGNE a maintenant la forme

|| 02 LIGNE OCCURS 5 TIMES INDEXED BY IND-1||.

L'état du fichier FICHPAYE reste le même que dans l'exemple 10.26.

Si, à la suite d'exécution du programme, le premier enregistrement du fichier FICHPAYE sera transféré dans la mémoire intérieure, l'état de mémoire intérieure deviendra

|| ATELIER ≡ ␣␣MONTAGE$_0$MOIS ≡ 01$_0$NOM-
PRENOM (1) ≡ ␣␣␣EVANS-JULES$_0$NET (1) ≡ 130.12$_0$
COMMENTAIRE (1) ≡ ␣␣␣␣␣␣␣␣␣␣$_0$NOM-
PRENOM (2) ≡ FOURIER-CLAUDE$_0$ NET (2) ≡ 100.00$_0$
COMMENTAIRE(2) = ␣␣␣␣␣␣␣␣␣␣°NOM-
PRENOM (3) ≡ ␣LANSON-NESTOR$_0$NET (3) ≡ 224.01$_0$
COMMENTAIRE (3) ≡ ␣␣␣␣␣␣␣␣␣␣$_0$NOM-
PRENOM (4) ≡ ␣␣␣HOUDON-JEAN$_0$NET(4) ≡ 114.00$_0$
COMMENTAIRE(4) ≡ ␣␣␣␣␣␣␣␣␣␣$_0$NOM-
PRENOM(5) ≡ ␣MURAT-JOACHIM$_0$NET (5) ≡ 098.50$_0$
COMMENTAIRE(5) ≡ ␣␣␣␣␣␣␣␣␣␣$_0$||.

En réunissant les états de mémoires extérieure et intérieure on aboutit à un état de mémoire, i.e. à une donnée traitée au cours de l'exécution du programme COBOL.

§ 10.8. Expressions utilisées en COBOL

Des expressions arithmétiques et logiques sont utilisées dans la division de traitement d'un programme COBOL. Elles sont analogues aux expressions simples de l'ALGOL. Formulons leurs définitions rigoureuses.

10.8.1. Expression arithmétique. La définition de l'expression arithmétique est récursive.

1) On appelle *primaire arithmétique* un nom-de-donnée, un nom-de-donnée-indicé, un littéral numérique non signé, une expression arithmétique entre parenthèses.

2) On appelle *facteur* soit un primaire arithmétique, soit un facteur suivi du symbole || ** || d'exponentiation suivi d'un primaire arithmétique.

3) On appelle *terme* soit un facteur, soit une suite se composant d'un terme suivi d'un symbole de multiplication ou de division (|| * || ou ||/||) suivi d'un facteur.

4) On appelle *expression arithmétique* soit un terme, soit un terme précédé d'un signe moins (|| —||) ou plus (|| +||), soit une suite se composant d'un primaire arithmétique, d'un signe plus ou moins et d'un terme.

Remarquons qu'en vertu de la définition donnée, les littéraux —2 et +2 sont des expressions arithmétiques et non pas des primaires arithmétiques.

La valeur de l'expression arithmétique est calculée dans le même ordre que dans l'ALGOL. L'hiérarchie des opérations arithmétiques est usuelle; l'élévation à une puissance est prioritaire sur toutes les autres opérations arithmétiques, ensuite viennent la multiplication et la division qui sont de même ordre hiérarchique, enfin l'addition et la soustraction qui sont de l'ordre hiérarchique le plus faible. Lorsque l'ordre d'une séquence d'opérations consécutives de même niveau hiérarchique n'est pas complètement précisé par des parenthèses, les opérations se font de gauche à droite. Si un opérande participe dans des opérations d'hiérarchies différentes, on commence par les opérations prioritaires; si un opérande représente une expression entre parenthèses, cette expression est calculée avant d'exécuter l'opération en question.

Exemple 10.28. Voici des expressions arithmétiques:

|| SOMME + 2 * (VALEUR-1 MOINS VALEUR-2)||

|| X + 2.5 * Y — 3 * (8 * X * * (Z — 4) + 1)||.

Dans la dernière expression on calcule d'abord 2.5 * Y; désignons le résultat par r_1; puis on calcule $X + r_1 = r_2$; puis $Z - 4 = r_3$; puis $X * * r_3 = r_4$; puis $8 * r_4 = r_5$; puis $r_5 + 1 = r_6$; puis $3 * r_6 = r_7$ et enfin $r_2 + r_6$ ce qui représente la valeur de l'expression.

10.8.2. Expression logique. Les expressions logiques les plus simples sont les relations. On considère dans le COBOL les *relations unaires* et les *relations binaires.*

Les relations unaires sont:

a) les *conditions de signe* dont chacune représente une suite se composant d'une expression arithmétique (en particulier, un nom-de-donnée), du mot || IS|| et de l'un des mots suivants: || POSITIVE|| ZERO|| NEGATIVE||;

b) les *conditions de classe* dont chacune représente un nom-de-donnée suivi des mots || IS NUMERIC|| ou || IS ALPHABETIC||;

c) la variable logique ou le *nom-de-condition* (voir le p. 10.6.2). Une telle variable est définie dans la section des données par une description de niveau 88. Une variable logique est un nom d'une valeur particulière d'une variable.

Une relation binaire est une notation qui se compose d'un premier membre de relation suivi d'un signe de relation (|| >|| <|| =|| GREATER THAN || LESS THAN|| EQUAL TO||) suivi d'un deuxième membre de relation. Le premier et le deuxième membre de relation sont des expressions arithmétiques (qui peuvent se réduire à un nom-de-donnée ou à un littéral numérique). Il n'est pas conseillé de mettre des littéraux numériques dans les deux membres, car la valeur logique de la relation sera dans ce cas constante.

Lorsque la vérification montre qu'une relation est satisfaite, on lui attribue la valeur logique « vrai »; dans le cas contraire elle prend la valeur « faux ».

Exemple 10.29. Donnons quelques exemples de relations.

Commençons par des relations binaires. Toute relation binaire peut être mise sous l'une des deux formes, en utilisant un symbole de relation ou les mots réservés ayant la même signification. Voici des expressions binaires :

	SALAIRE EQUAL TO 200	
	REVENU EQUAL TO DEPENSE	
	PRIX LESS THEN 50	
	DELAI GREATER THAN 12	

	SALAIRE-200	
	REVENU-DEPENSE	
	PRIX <50	
	DELAI >12	
	(X + Y) * Z >** 2 +
+ 3 ||.

Voici des relations unaires :

	ACCROISSEMENT IS POSITIVE	
	X PLUS Y IS ZERO	
	ACCROISSEMENT IS NEGATIVE	
	NOM IS NUMERIC
	NOM IS ALPHABETIC	
	FEVRIER||.

La dernière des relations unaires est le nom d'une valeur de la donnée MOIS. Lorsque sont traitées des données concernant le mois de février, la relation unaire FEVRIER sera considérée comme satisfaite.

Remarquons qu'une relation unaire peut parfois être remplacée par une relation binaire contenant des littéraux. Ainsi, la première, la deuxième, la troisième et la dernière des relations unaires ci-données sont respectivement équivalentes aux relations binaires || ACCROISSEMENT >0|| X PLUS Y = 0|| ACCROISSEMENT <0|| || MOIS-02||. Commentons la dernière relation binaire. Le nom-de-donnée MOIS a pour valeurs les numéros des mois 01, 02, . . ., 11, 12. Le FEVRIER est un nom de la valeur 02 de la donnée MOIS.

Signalons spécialement que les relations binaires définies plus haut n'ont un sens que lorsque les deux membres d'une relation sont compatibles en classe. La classe d'une donnée élémentaire est définie par sa chaîne PICTURE. Les grandeurs à comparer peuvent être: 1) numériques; 2) alphabétiques, 3) alphanumériques, 4) numériques éditées. Les grandeurs numériques sont compatibles avec les grandeurs numériques, les grandeurs alphabétiques avec les grandeurs alphabétiques ou alphanumériques, les quantités alphanumériques avec les alphabétiques ou les alphanumériques, les grandeurs numériques éditées avec seulement les grandeurs numériques éditées.

La comparaison de grandeurs numériques se fait d'après les règles de comparaison de nombres. Dans les autres cas, la comparaison de grandeurs se fait comme comparaison de leurs positions relatives dans la séquence de classement des caractères du code EBCDIC définie pour l'ordinateur considéré.

EXEMPLE 10.30. Pour IBM, un caractère dépasse un autre si, en considérant les codes binaires de ces caractères comme nombres entiers en binaire, on constate que la valeur du premier code est supérieure à celle du deuxième. La comparaison se fait caractère par caractère, de gauche à droite. Dès qu'on rencontre une paire de caractères différents, ces deux caractères sont comparés pour connaître leur position relative dans la séquence de classement du code EBCDIC. L'opérande contenant le caractère dont la position est la plus élevée dans la séquence est le plus grand des deux opérandes.

Ainsi, REVENU se trouve supérieur à RETOUR, puisque V est supérieur à T (leurs caractères EBCDIC sont respectivement 01011110 et 00111110), TABLE se trouve inférieur à TABLES.

On peut comparer des données élémentaires (comme c'est décrit plus haut), ainsi que des groupes de données. Les groupes ne sont comparables que si leurs « tailles » coïncident (i.e. les zones de mémoire qui leur sont réservées contiennent un même nombre de cases).

Les groupes se comparent par cases. Il faut en tenir compte, lorsque la relation donnée contient des données décrites dans la section des mémoires de travail. De telles données ne doivent pas redéfinir les données d'une taille sensiblement plus grande.

A partir des relations, on peut former des expressions logiques plus compliquées à l'aide de connecteurs logiques. Formulons une définition récursive de l'expression logique.

1) On appelle *primaire logique* une relation (unaire ou binaire) ou une expression logique entre parenthèses.

2) On appelle *secondaire logique* soit un primaire logique, soit un primaire logique précédé du mot ||NOT||.

3) On appelle *facteur logique* soit un secondaire logique, soit plusieurs secondaires logiques liés par des mots || AND ||.

4) On appelle *expression logique* soit un facteur logique, soit plusieurs facteurs logiques liés par des mots || OR ||.

Exemple 10.31. Toutes les relations de l'exemple 10.29 sont des primaires logiques. Les notations suivantes sont des expressions logiques :

|| NOT FEVRIER || NOT ACCROISSEMENT IS POSITIVE ||
|| NOT X + Y = Z ||
|| NOT X + Y EQUAL TO Z || ACCROISSEMENT IS POSITIVE AND FEVRIER || NOM IS ALPHABETIC AND X PLUS Y GREATER THAN Y ** 2 ||.

Pour calculer les expressions logiques, on adopte l'hiérarchie suivante des opérations (par ordre décroissant) : calcul d'une expression arithmétique, signes de relations (unaires et binaires), symbole NOT, symbole AND, symbole OR.

Lorsque l'ordre d'exécution des opérations logiques n'est pas déterminé par des parenthèses, c'est ainsi qu'on calcule une expression logique :

a) on calcule toutes les expressions arithmétiques de gauche à droite ;

b) on calcule de même les valeurs des relations (qui sont VRAI ou FAUX) ;

c) en allant de gauche à droite on trouve le premier symbole d'une opération logique dont les opérandes ne sont pas opérandes d'une opération logique prioritaire. On exécute cette première opération et remplace son symbole et ses opérandes par le résultat obtenu. On répète b) jusqu'à aboutir à la valeur de l'expression logique donnée.

Si l'ordre de calcul d'une expression logique ne change pas en supprimant les parenthèses qui déterminent l'ordre des opérations logiques, on dit que l'expression logique permet la suppression de parenthèses. Par exemple, l'expression logique

|| (((X GREATER THAN Y) AND (Y LESS THAN C)) OR (X IS ALPHABETIC)) ||

est équivalente à l'expression

|| X GREATER THAN Y AND Y LESS THAN C OR X IS ALPHABETIC ||.

Par contre, l'expression

|| (((X GREATER THAN Y) OR (Y LESS THAN C)) AND (X IS ALPHABETIC))||

ne permet que la suppression des parenthèses extérieures et des parenthèses contenant les relations, ce qui fournit

|| (X GREATER THAN Y OR Y LESS THAN C) AND X IS ALPHABETIC||.

On ne peut plus supprimer les parenthèses.

Il est permis dans le COBOL de permuter et d'omettre certains mots afin de rendre les notations plus lisibles. Par exemple, on peut remplacer || NOT A LESS THAN B|| par || A NOT LESS THAN B||; || A LESS THAN B AND A LESS THAN C|| par || A LESS THAN B AND C||; || A LESS THAN B OR A EQUAL TO C|| par || A LESS THAN B OR EQUAL TO C||.

Dans les transformations d'expressions logiques on se guidera des règles suivantes.

0°. On vérifie si l'expression donnée permet la suppression de toutes les parenthèses déterminant l'ordre d'opérations logiques; si oui on passe au point suivant; si non la règle n'est pas applicable.

1° On barre dans l'expression logique les paires de parenthèses qui contiennent des relations isolées (binaires ou unaires).

2°. On examine l'expression en cherchant les relations précédées du symbole NOT, on y permute NOT et le premier membre (dans une relation unaire ce sera son unique membre). Par la suite, la notation formée par NOT et le symbole de relation est considérée comme symbole de relation.

3°. On choisit deux relations liées par un symbole AND ou OR. On barre dans la deuxième son premier ou ses premier et deuxième éléments *) s'ils sont identiques respectivement au premier ou aux premier et deuxième éléments de la première relation. Si le résultat de cette opération contient une suite se composant d'anciens deuxièmes éléments liés par les symboles AND ou OR, on considère cette suite comme un nouveau deuxième membre de la relation.

Si le résultat mentionné contient une suite formée par le symbole de la première relation, son deuxième membre, le symbole AND ou OR et le symbole de l'ancienne deuxième relation, on considérera cette suite comme un nouveau symbole de relation.

4°. Si l'on veut continuer, on reprend le processus dès le p. 1°.

---

*) Nous appelons éléments d'une relation ses membres et le symbole de la relation. Ainsi, dans la relation $A > B$, les éléments sont A (premier membre), B (deuxième membre) et $>$ (symbole de relation).

EXEMPLE 10.32. Soit une relation logique || (X IS NUMERIC AND X GREATER THAN 2) OR NOT X LESS THAN Z||.

En permutant NOT et X dans la dernière relation et barrant le deuxième X entre parenthèses (conformément aux pp. 1°, 2°, 3°) nous obtenons || (X IS NUMERIC AND GREATER THAN 2) OR X NOT LESS THAN Z||. Dans cette relation IS NUMERIC et GREATER THAN, ainsi que NOT LESS THAN sont considérés comme symboles de relations. Poursuivons la simplification. En supprimant les parenthèses (i.e. en exécutant le p. 1°) nous obtenons ||X IS NUMERIC AND GREATER THAN 2 OR X NOT LESS THAN Z||.

En barrant X dans la deuxième relation de la dernière expression logique nous avons définitivement || X IS NUMERIC AND GREATER THAN 2 OR NOT LESS THAN Z||.

Dans cette expression logique, la suite des mots IS NUMERIC AND GREATER THAN 2 OR NOT LESS THAN est considérée comme symbole de relation (si l'on tient à poursuivre le processus). Dans le présent cas la réduction n'est plus possible.

L'expression logique

|| ((X IS NUMERIC AND X GREATER THAN 2) OR NOT X LESS THAN Y) AND ((X IS NUMERIC AND X GREATER THAN 2) OR NOT X LESS THAN Z)||

peut d'abord être simplifiée comme suit:

|| (X IS NUMERIC AND GREATER THAN 2 OR NOT LESS THAN Y) AND (X IS NUMERIC AND GREATER THAN 2 OR NOT LESS THAN Z)||,

puis mise sous la forme

|| X IS NUMERIC AND GREATER THAN 2 OR NOT LESS THAN Y AND Z||.

§ 10.9. Division de traitement du programme COBOL

La division de traitement, comme le dit son nom, contient la description du processus de traitement de données. De plus, on y décrit les entrées-sorties d'enregistrements. La division commence par l'en-tête

|| PROCEDURE DIVISION.||.

10.9.1. Structure de la division de traitement. Les éléments principaux de cette division sont les instructions, ou ordres.

Une ou plusieurs instructions suivies d'un point || .|| constituent une *phrase*. Il est permis de séparer deux instructions d'une phrase par une virgule.

Une ou plusieurs phrases peuvent être réunies en un *paragraphe*. Chaque paragraphe doit avoir son *nom* qui est mis avant la première phrase du paragraphe et suivi d'un point. Un nom de paragraphe est un nom non réservé, en particulier un entier non signé.

Un ou plusieurs paragraphes peuvent être réunis en une section. Chaque section a son en-tête composé du mot ||SECTION|| suivi d'un mot non réservé qui représente le nom de la section, puis d'un point. Une section est la plus grande subdivision de la division de la procédure. Elle contient tous les paragraphes qui se trouvent entre son nom et celui de la section suivante ou la fin du programme. Tout à fait de même, à un paragraphe appartiennent toutes les instructions qui se trouvent entre son nom et celui du paragraphe suivant ou d'une section. Pourtant, bien que le découpage de la suite de paragraphes en parties appelées sections soit arbitraire, la segmentation d'une suite de phrases en paragraphes dépend du type d'instructions. Les instructions de la PROCEDURE DIVISION sont exécutées dans l'ordre de leur notation, sauf dans le cas où une modification de l'ordre séquentiel est prévue. Alors, pour opérer cette modification, on indique dans l'instruction le nom du paragraphe ou de la section qui est à exécuter. Dans le premier cas on passe à la première instruction du paragraphe de nom indiqué, dans le deuxième, à la première instruction du premier paragraphe de la section mentionnée. Il est possible d'éviter de mentionner ce nom de section en appelant le premier de ses paragraphes. La méthode de définition de l'instruction PERFORM (voir le p. 10.11.2) implique aussi le découpage de phrases en paragraphes.

Les noms de sections et de paragraphes de la PROCEDURE DIVISION jouent, pour les transferts de la commande, le rôle d'étiquettes de l'ALGOL. On appelle ces noms par le terme commun noms-de-traitement.

10.9.2. Instructions. Il existe deux types d'instructions du COBOL : les *instructions impératives* et les *instructions conditionnelles*. Les phrases ne contenant que des instructions impératives sont dites *impératives*.

Une phrase qui contient une instruction conditionnelle s'appelle *phrase conditionnelle*. Une phrase conditionnelle peut contenir des instructions impératives (en particulier, n'en pas contenir du tout) et une seule instruction conditionnelle qui est la dernière dans la phrase.

Chaque instruction commence par un *verbe* qui est un mot réservé. On appelle par ces mots les instructions correspondantes du COBOL.

Les principales instructions impératives sont :

1) les instructions arithmériques : COMPUTE (calculer), ADD (additionner), SUBTRACT (soustraire), MULTIPLY (multiplier), DIVIDE (diviser) ;

2) les instructions de contrôle : GO TO (aller à), ALTER (modifier), PERFORM (exécuter), EXIT (sortir), STOP (arrêter) ;
3) les instructions d'entrée-sortie : OPEN (ouvrir), SEEK (chercher), CLOSE (fermer), READ (lire), WRITE (écrire), ACCEPT (accepter), DISPLAY (sortir) ;
4) les instructions de transfert des données : MOVE (mettre), EXAMINE (examiner);
5) les instructions de tri : SORT (classer) ;
6) les instructions d'appel des sous-programmes : CALL, ENTRY, EXIT PROGRAM.

Une instruction GO TO ou STOP ne peut que terminer la phrase dont elle fait partie (même si cette instruction est contenue dans une instruction conditionnelle).

Les principales instructions conditionnelles sont :
1) l'option ON SIZE ERROR des ordres arithmétiques ;
2) la forme AT END de l'instruction READ ;
3) les instructions IF.

Toutes les instructions impératives sont naturellement simples ; les instructions conditionnelles sont composées (contiennent d'autres instructions dans son format).

§ 10.10. Instructions arithmétiques

10.10.1. Instructions arithmétiques impératives. Une instruction COMPUTE représente une suite se composant du verbe || COMPUTE || d'un nom-de-donnée élémentaire, du signe d'égalité et d'une expression arithmétique. Le signe d'égalité est considéré comme le signe d'affectation. Le nom-de-donnée peut être suivi du mot || ROUNDED ||.

L'instruction COMPUTE calcule l'expression arithmétique qui en fait partie et attribue sa valeur au nom-de-donnée qui précède le signe d'égalité. Si après alignement sur la marque décimale, le nombre de décimales du résultat est plus grand que le nombre de décimales prévues dans la donnée-résultat, une troncature est effectuée. Si la clause ROUNDED est spécifiée, la valeur de la décimale la moins significative de la donnée-résultat est augmentée de 1 lorsque le chiffre le plus significatif de la partie abandonnée est supérieur ou égal à 5.

EXEMPLE 10.33. Voici des instructions COMPUTE :

|| COMPUTE MONTANT = HEURE * TAUX ||

|| COMPUTE POIDS-SPECIFIQUE ROUNDED = POIDS/VOLUME ||

|| COMPUTE P ROUNDED = Q/V ||.

Les instructions arithmétiques ADD, SUBTRACT, MULTIPLY, DIVIDE représentent chacune une suite formée par: 1) un verbe; 2) une liste de noms-de-donnée élémentaires (dont le nombre sera désigné par n) qui contient avant le dernier nom-de-donnée l'un des mots || TO||, || FROM||, || BY||, || BY|| ou || INTO|| respectivement pour les instructions ADD, SUBTRACT, MULTIPLY, DIVIDE, 3) le mot || GIVING|| suivi d'un nom-de-donnée élémentaire (cette option est facultative; si l'on choisit l'option GIVING dans l'instruction ADD, on n'écrit pas le mot || TO||), 4) le mot facultatif || ROUNDED||.

La liste de noms-de-donnée élémentaires représente une suite de noms-de-donnée élémentaires que l'on peut séparer par des virgules.

L'exécution d'une instruction ADD ou SUBTRACT consiste en ceci: a) on additionne $n - 1$ premières données de la liste; b) on effectue sur la somme obtenue et la n-ième donnée élémentaire de la liste l'opération spécifiée par le verbe; c) le résultat de cette dernière est affecté au n-ième nom-de-donnée, si GIVING fait défaut, et au nom-de-donnée suivant le mot GIVING, si ce dernier est spécifié. L'option ROUNDED prévoit l'arrondissement, de même que pour l'instruction COMPUTE.

En cas d'instructions MULTIPLY ou DIVIDE, la liste de données élémentaires contient deux noms-de-donnée ($n = 2$). L'exécution de l'instruction se déroule de même que pour les verbes ADD et SUBTRACT, à ceci près qu'on ne fait pas l'addition des $n - 1$ premières données, puisque leur somme est égale à la première donnée élémentaire.

Dans les instructions arithmétiques considérées, un nom-de-donnée peut être remplacé par un littéral numérique, si ce n'est pas un nom de résultat.

Remarquons que, si un dépassement de capacité se produit au cours de l'exécution d'une instruction arithmétique impérative, les poids forts du résultat seront perdus. Pour y remédier, on peut se servir d'instructions arithmétiques conditionnelles (voir le p. 10.10.2).

EXEMPLE 10.34. Les textes suivants sont des instructions arithmétiques:

	ADD X Y TO Z	
	ADD X Y TO Z ROUNDED	
	ADD X Y Z GIVING U	
	SUBTRACT IMPOT FROM REVENU	
	MULTIPLY X BY Y GIVING Z ROUNDED	
	DIVIDE X INTO Y	
	DIVIDE X BY Y GIVING Z ROUNDED||.

10.10.2. Instructions arithmétiques conditionnelles. On obtient une instruction arithmétique conditionnelle en faisant suivre une instruction arithmétique non conditionnelle des mots || ON SIZE ERROR || suivis d'une instruction impérative quelconque. L'emploi d'une instruction arithmétique conditionnelle permet au programmeur soit de prévoir la transmission du signal d'erreur de dimension à la console, soit d'éliminer cette erreur par programme.

Exemple 10.35. Les textes suivants sont des instructions arithmétiques conditionnelles :

|| COMPUTE X ROUNDED = (X + Y) * Z ** W
ON SIZE ERROR DISPLAY 'PANNE' ||
|| MULTIPLY X BY Y GIVING Z ROUNDED
ON SIZE ERROR STOP 'PANNE' ||
|| SUBTRACT X FROM Y GIVING Z
ON SIZE ERROR GO TO CORRECTION ||.

En cas d'erreur de dimension, la première des instructions données fournira au pupitre le mot PANNE, l'exécution du programme ne sera pas interrompue (voir p. 10.12.3). La deuxième instruction fournira le même mot au pupitre et arrêtera la machine (voir le p. 10.11.1). La troisième instruction, en cas d'erreur de dimension, passera le contrôle au paragraphe nommé CORRECTION (voir le p. 10.11.1).

§ 10.11. Instructions de commande de séquence

On appelle *procédure* n'importe quelle partie de la PROCEDURE DIVISION ayant un en-tête. Ainsi, les procédures dans un programme COBOL sont les sections et les paragraphes de sa PROCEDURE DIVISION. Nous entendons par « nom-de-traitement » un nom de paragraphe ou de section.

Le terme « procédure » du COBOL n'est pas à confondre avec le même terme de l'ALGOL.

Nous avons déjà dit qu'on classe parmi les instructions de contrôle les instructions de branchement inconditionnel et conditionnel (GO TO), l'instruction qui modifie une instruction GO TO (ALTER), l'instruction de boucle (PERFORM), l'instruction qui fournit un successeur commun à toutes les sorties d'une boucle (EXIT) et l'instruction d'arrêt (STOP).

10.11.1. Instructions GO TO, ALTER et STOP. L'instruction GO TO inconditionnel se compose des mots || GO TO || suivis d'un nom-de-traitement. Elle passe la commande à la première instruc-

tion de la section ou du paragraphe dont le nom figure dans l'instruction considérée.

L'instruction GO TO sélectif représente une suite se composant des mots || GO TO||, d'une liste de noms-de-traitement, des mots || DEPENDING ON|| et d'un nom-de-donnée élémentaire. Cette donnée élémentaire doit être décrite dans la DATA DIVISION (dans la clause PICTURE qui lui correspond) en tant que donnée numérique sans point. L'exécution de l'instruction GO TO selectif commence par ce qu'on choisit dans la liste de noms-de-traitement celui dont le numéro d'ordre (à compter du début de la liste) est égal à la valeur de la donnée élémentaire mentionnée. Ensuite la commande passe à la première instruction de la procédure choisie.

L'instruction GO TO doit être toujours la dernière dans la phrase.

Exemple 10.36. GO TO inconditionnel:

|| GO TO MULTIPLICATION-DE-MATRICES||.

GO TO selectif:

|| GO TO CALCUL CLASSEMENT DEPENDING
ON VOLUME-DE-PRODUCTION||.

L'instruction ALTER représente le mot || ALTER|| suivi d'un nom-de-traitement, puis du mot || TO|| (ou des mots || TO PROCEED TO||), et enfin d'une liste de noms-de-traitement. Elle sert à modifier en cours de programme une instruction GO TO, qui doit être l'unique dans la procédure dont le nom suit immédiatement le mot ALTER. L'instruction GO TO à modifier doit contenir une liste de noms-de-traitement comprenant autant de procédures qu'il y en a dans la liste de l'instruction ALTER. L'instruction ALTER trouve le GO TO à modifier et remplace sa liste de procédures par une autre.

Exemple 10.37. Voici deux instructions ALTER:

|| ALTER A-2 TO PROCEED TO CALCUL||

|| ALTER A-35 VERIFICATION CALCUL CLASSEMENT||.

Une instruction STOP représente le mot || STOP|| suivi d'un littéral ou du mot || RUN||.

Si l'option « littéral » est utilisée, l'instruction provoque la sortie d'un code message concernant le littéral spécifié, puis l'arrêt du programme. On peut reprendre l'exécution du programme en composant sur le pupitre le code message. Dans l'option RUN, l'exécution du programme s'arrête et le contrôle passe au système d'exploitation.

Une instruction STOP doit être la dernière dans sa phrase (qu'elle soit impérative ou conditionnelle).

10.11.2. Instructions PERFORM et EXIT. Décrivons trois options principales de l'instruction PERFORM qui permet d'exécuter une séquence de traitement un nombre de fois dépendant du conditionnement spécifié.

La première option assure l'exécution d'une séquence de traitement un nombre donné de fois.

La deuxième option provoque la répétition d'une séquence de traitement jusqu'à ce que, avant une nouvelle exécution de la séquence, une expression logique donnée ne soit vraie (si cette expression est vraie avant la première exécution de la séquence, l'instruction est ineffective).

La troisième option provoque l'exécution d'une séquence de traitement chaque fois que varient les valeurs d'une ou de deux grandeurs pour lesquelles sont données la valeur initiale, l'incrément (pas) et une condition qui limite le nombre d'exécutions.

Dans la première option l'instruction PERFORM comporte:
le mot || PERFORM || suivi d'un nom-de-traitement;

le mot || THRU || suivi d'un autre nom-de-traitement; cette clause s'utilise dans le cas où la séquence de traitement contient plus d'une procédure, la procédure qui y est spécifiée est la dernière dans la série;

un entier ou un nom-de-donnée suivi du mot || TIMES ||.

L'option décrite de l'instruction PERFORM transmet la commande à la première instruction de la procédure spécifiée après le mot PERFORM. Après l'exécution de sa dernière instruction (ou après l'exécution de la dernière instruction de la procédure nommée à la suite de THRU si THRU est spécifié) la séquence s'exécute une nouvelle fois ou, si elle est déjà exécutée le nombre donné de fois, la commande passe à l'instruction qui suit immédiatement l'instruction PERFORM considérée. Il est recommandé de constituer une séquence de traitement d'une telle façon qu'après le nombre donné de répétitions, l'instruction qui suit le PERFORM puisse prendre la commande. Il faut pour cela que la séquence en question ne contienne à son intérieur aucune instruction GO TO menant à l'extérieur.

Remarquons que, si le mot TIMES est précédé d'un nom-de-donnée, le nombre d'exécutions de la séquence de traitement est déterminé avant la première exécution. Par conséquent, une modification de la valeur de cette donnée au cours d'exécution est sans effet sur le nombre mentionné.

La deuxième option de l'instruction PERFORM comporte:
le mot || PERFORM || suivi d'un nom-de-traitement;

le mot || THRU || suivi d'un nom-de-traitement (cette clause est spécifiée si la séquence de traitement contient plus d'une procédure);

le mot || UNTIL || suivi d'une expression logique.

Cette option travaille de même que la précédente, à ceci près qu'avant chaque exécution de la séquence de traitement la valeur de l'expression logique est examinée. Si elle s'avère vraie, l'exécution de la séquence ne se répète plus, et la commande passe à l'instruction qui suit immédiatement le PERFORM.

La troisième option de l'instruction PERFORM comporte:

le mot || PERFORM || et un nom-de-traitement;

le mot || THRU || et un nom-de-traitement (la clause est spécifiée si la séquence de traitement contient plus d'une procédure||;

le mot || VARYING || suivi d'un nom-de-donnée élémentaire;

le mot || FROM || suivi d'un nom-de-donnée ou d'un littéral numérique;

le mot || BY || suivi d'un nom-de donnée ou d'un littéral numérique;

le mot || UNTIL || suivi d'une expression logique.

Dans cette option l'instruction PERFORM s'exécute comme suit:

a) la valeur initiale du paramètre de boucle (du nom-de-donnée suivant VARYING) est rendue égale à la valeur suivant FROM;

b) la valeur de l'expression logique est examinée. Si l'expression est fausse, la séquence de traitement s'exécute. Dans le cas contraire la commande passe à l'instruction suivant PERFORM;

c) après l'exécution de la séquence de traitement la valeur de la variable de contrôle est incrémentée de la valeur indiquée derrière BY, tout le processus se répète à commencer par b).

Si l'on désigne par i la variable de contrôle, par i_0 sa valeur initiale, par h l'incrément, alors le processus de répétitions de la séquence de traitement par l'instruction PERFORM peut être écrit en YALS comme suit:

$$i:=i_0;\quad \underset{2}{\lrcorner}\, P \overset{1}{\ulcorner} Ri:=i+h;\ \underset{2}{\llcorner},$$

où P est l'expression logique figurant dans le PERFORM, R la séquence de traitement, par le signe $\overset{1}{\ulcorner}$ est exprimé le transfert de commande à l'instruction qui suit immédiatement le PERFORM.

La troisième option du PERFORM admet l'utilisation de deux ou de trois variables de contrôle. On associe alors à chaque paramètre les clauses correspondantes, en séparant les clauses concernant les différents paramètres par le mot || AFTER ||. En cas de deux paramètres, l'exécution de l'instruction PERFORM aura la forme:

$$i:=i_0;\ \underset{4}{\lrcorner}\, P_1 \overset{1}{\ulcorner}\ j:=j_0;\ \underset{3}{\lrcorner}\, P_2 \overset{2}{\ulcorner} Rj:=j+h_2;\ \underset{3}{\llcorner}\,\underset{2}{\lrcorner}\, i:=i+h_1;\ \underset{4}{\llcorner}$$

En cas de trois paramètres ce sera :

$$i := i_0;\ \underset{6}{\lrcorner}\ P_1 \overset{1}{\ulcorner}\ j := j_0;\ \underset{5}{\lrcorner}\ P_2 \overset{2}{\ulcorner}\ k := k_0;\ \underset{4}{\lrcorner}\ P_3 \overset{3}{\ulcorner}\ R\ k := k + h_3;$$

$$\underset{4}{\llcorner}\ \underset{3}{\lrcorner}\ j := j + h_2;\ \underset{5}{\llcorner}\ \underset{2}{\lrcorner}\ i := i + h_1;\ \underset{6}{\llcorner}$$

Dans les deux dernières notations, i, j, k sont des variables de contrôle, i_0, j_0, k_0 leurs valeurs initiales, h_1, h_2, h_3 les incréments, P_1, P_2, P_3 les expressions logiques correspondantes. Le signe $\overset{1}{\ulcorner}$ rend toujours la commande à l'instruction qui suit immédiatement le PERFORM.

Une séquence de traitement spécifiée dans une instruction PERFORM peut contenir une autre instruction PERFORM avec une autre séquence de traitement. Alors la deuxième séquence ou bien ne doit pas avoir de procédure commune avec la première, ou bien elle doit faire partie de cette première séquence sans avoir avec elle la procédure finale commune. Cela veut dire, en particulier, qu'une instruction PERFORM ne peut être contenue dans une procédure de la séquence de traitement qu'elle nomme.

EXEMPLE 10.38. Donnons des exemples de toutes les options de l'instruction PERFORM.

Première option.

|| PERFORM PARAGRAPHE-2 25 TIMES||

|| PERFORM PARAGRAPHE-2 NOMBRE-D-USINES TIMES||

|| PERFORM PARAGRAPHE-2 THRU PARAGRAPHE-10 42 TIMES||.

Deuxième option.

|| PERFORM CALCUL UNTIL TOTAL > 1000||

|| PERFORM CALCUL THRU CORRECTION UNTIL TOTAL > 1000||.

Troisième option.

|| PERFORM CALCUL THRU CORRECTION VARYING INDICE-1 FROM 1BY 2 UNTIL INDICE-1 IS EQUAL TO 11 AFTER
REVENU FROM REVENU-65 BY TAUX UNTIL REVENU-73||.

L'instruction EXIT est formée par l'unique mot || EXIT|| et doit constituer à elle seule un paragraphe, qui peut figurer en tant que la dernière procédure d'une séquence de traitement indiquée dans une instruction PERFORM. Au cours de l'exécution de l'instruc-

tion PERFORM, toute sortie de la séquence de traitement est alors remplacée par un transfert de commande à l'instruction EXIT. Chaque instruction EXIT ne fait que rendre la commande à l'instruction qui la suit immédiatement.

Une instruction EXIT n'est au fait qu'un complément à l'instruction PERFORM où après le mot UNTIL est nommé le paragraphe qui la contient.

§ 10.12. Instructions d'entrée-sortie

10.12.1. Instructions OPEN et CLOSE. L'instruction OPEN rend possible l'exécution d'une instruction READ ou WRITE, i.e. prépare les fichiers et les supports où ils sont stockés à l'entrée ou à la sortie des données. On dit encore que l'exécution d'une instruction OPEN permet d'initialiser le traitement d'un ou de plusieurs fichiers. Une instruction OPEN contient:

le mot || OPEN ||;

l'un des mots || INPUT ||, || OUTPUT ||, || I-O || suivi d'un nom-de-fichier;

l'une des clauses facultatives || REVERSED ||, || WITH NO REWIND || pour l'option INPUT, ou la clause facultative || || WITH NO REWIND || pour l'option OUTPUT.

Une instruction OPEN provoque l'exécution des procédures de contrôle des étiquettes pour les fichiers en entrée (INPUT et I-O), ou d'écriture des étiquettes pour les fichiers en sortie (OUTPUT).

L'option NO REWIND ne s'applique qu'aux unités physiques à bande magnétique.

L'option REVERSED s'applique seulement à des fichiers assignés à des unités de lecture admettant la « lecture arrière ».

Exemple 10.39. Voici des instructions OPEN:

|| OPEN INPUT 32M1 WITH NO REWIND ||
|| OPEN OUTUT A2B REVERSED ||
|| OPEN I-O FICHIER-3 ||.

L'instruction CLOSE sert à terminer le traitement d'un ou de plusieurs fichiers. Elle comprend:

le mot || CLOSE ||;

une liste de fichiers à fermer. Chaque élément de la liste est un nom-de-fichier suivi du mot || WITH || et de l'une des notations || NO REWIND || et || LOCK ||.

L'instruction CLOSE ne peut être spécifiée que si le fichier est ouvert. L'option LOCK provoque la fermeture du fichier et la libération de l'unité physique correspondante. L'option NO REWIND permet de ne pas rebobiner le fichier (s'il est enregistré sur bande).

Exemple 10.40. Voici une instruction CLOSE:

|| CLOSE M25 WITH LOCK M31 WITH NO REWIND M42||.

10.12.2. Instructions READ et WRITE. L'instruction READ introduit dans la mémoire principale un bloc du fichier qu'elle nomme. Cette instruction comporte dans l'ordre: le mot || READ||, un nom-de-fichier, le mot || RECORD||, la clause facultative formée du mot || INTO|| suivi d'un nom-de-donnée, les mots || AT END||, une instruction impérative.

L'instruction READ est conditionnelle. On a déjà dit (10.6.3) qu'un bloc peut contenir plusieurs articles, en particulier un seul. Si le bloc « lu » n'est pas le dernier dans le fichier, l'instruction READ passe la commande à la phrase qui la suit (dans sa phrase elle est la dernière, étant conditionnelle); et si le bloc « lu » est le dernier dans le fichier, l'instruction impérative contenue dans le READ s'exécute. Une instruction READ ne s'applique qu'à un fichier ouvert par une instruction OPEN INPUT ou OPEN I-O.

L'option INTO nom-de-donnée équivaut à une instruction READ et une instruction MOVE. Le nom-de-donnée est le nom d'une zone de mémoire décrite en WORKING-STORAGE SECTION ou d'un article d'un fichier de sortie préalablement ouvert. Lorsque cette option est utilisée, l'enregistrement en cours sera déplacé dans la zone nom-de-donnée suivant les règles de l'action de l'instruction MOVE.

Exemple 10.41. Voici une instruction READ qui prévoit la fermeture d'un fichier, après la lecture, et la libération de l'unité où il est stocké:

|| READ FICH-2 AT END CLOSE FICH-2 WITH LOCK||.

L'instruction WRITE permet de transférer un enregistrement de la mémoire principale dans un fichier en sortie. Celui-ci doit donc être ouvert par une instruction OUTPUT ou I-O.

L'instruction WRITE contient (nous n'en décrivons que quelques options principales):

le mot || WRITE|| suivi d'un nom-d'article;

la clause facultative formée par le mot || FROM|| suivi d'un nom-de-donnée;

les mots || BEFORE ADVANCING|| ou || AFTER ADVANCING||;

un nom-de-donnée suivi du mot || LINES|| ou un entier suivi de || LINES||.

La forme WRITE ... $\left\{\begin{matrix}\text{BEFORE}\\ \text{AFTER}\end{matrix}\right\}$ ADVANCING ...

n'est utilisée que lorsqu'un fichier est préparé pour l'impression (l'information correspondante figure dans ENVIRONMENT DIVI-

SION), et même dans ce cas on peut ne rien dire sur le déplacement de papier. Le mot ADVANCING désigne l'avance du papier avant (AFTER) ou après (BEFORE) l'impression. Le nombre de lignes sautées est spécifié soit directement par l'entier précédant le mot LINES, soit par le nom-de-donnée précédant ce mot. Dans le cas d'un nom-de-donnée, il doit représenter une zone contenant un entier positif inférieur à 100.

Lorsque l'option FROM est utilisée, le WRITE est équivalent à une instruction MOVE nom-de-donnée TO nom-d'article, suivie d'une instruction WRITE nom-d'article (voir le p. 10.13.1.)

Soulignons que l'instruction WRITE ne contient aucune information sur le fichier où sera placé l'article nommé dans le WRITE. Par conséquent, il ne doit pas y avoir plusieurs fichiers en sortie simultanément ouverts.

Exemple 10.42. Voici des instructions WRITE:

|| WRITE NOM AFTER ADVANCING 2 LINES||

|| WRITE ETAT-DE-SALAIRES||.

10.12.3. Instructions ACCEPT et DISPLAY. L'instruction ACCEPT se compose du mot || ACCEPT|| suivi d'un nom-de-donnée suivi des mots facultatifs || FROM CONSOLE||. Cette instruction permet l'introduction de données de faible volume à partir de la machine à écrire du pupitre si FROM CONSOLE est spécifié, ou à partir de l'unité standard d'entrée du système (SYSIN) dans le cas contraire.

Le nom-de-donnée peut être soit un élément de groupe de longueur fixe, soit un élément simple alphabétique, alphanumérique ou décimal étendu. Un enregistrement logique est lu et le nombre approprié de caractères est déplacé vers la zone réservée pour le nom-de-donnée.

L'instruction DISPLAY sert à écrire un article sur une unité de sortie. Elle contient:

le mot || DISPLAY||;

une liste d'informations à sortir;

le mot || UPON|| suivi d'un nom d'unité de sortie spécifié dans le paragraphe SPECIAL-NAMES de l'ENVIRONMENT DIVISION; cette clause est facultative; si elle n'est pas spécifiée, il se produit la sortie sur la machine à écrire du pupitre.

Un élément de la liste de sortie représente un nom-de-donnée élémentaire ou un littéral (sans signe).

Exemple 10.43. Les notations suivantes sont des instructions ACCEPT:

|| ACCEPT NOM-OPERATION||

|| ACCEPT QUE-FAIRE FROM CONSOLE||.

Voici des exemples de l'instruction DISPLAY:

|| DISPLAY 25.31 NOM ZEROS||
|| DISPLAY FRAIS UPON CONSOLE||
|| DISPLAY ERREUR-DIMENSION||.

§ 10.13. Instructions de transfert des données

10.13.1. Instructions MOVE et EXAMINE. L instruction MOVE se compose du mot || MOVE|| suivi d'un nom-de-donnée ou d'un littéral, puis du mot || TO|| et enfin d'une liste de noms-de-donnée. L'exécution de cette instruction consiste en le transfert du contenu de la zone de mémoire principale désignée par le premier nom-de-donnée (ou par le littéral) dans une ou plusieurs zones réceptrices désignées par des noms-de-donnée de la liste (avec la conservation de la donnée transférée dans la zone initiale).

Si les données d'origine et réceptrice sont élémentaires, alors la donnée déplacée est ajustée au cours du transfert pour correspondre à la donnée réceptrice (des tronquages et des complétions par des blancs ou par d'autres symboles sont possibles). Il se peut que la donnée réceptrice soit une donnée éditée.

Si les données émittrice et réceptrice sont des éléments de groupe, leurs constructions doivent être identiques.

Exemple 10.44. Voici des instructions MOVE:

|| MOVE DONNEE TO ZONE-1 ZONE-2||
|| MOVE PRIX TO COUT||
|| MOVE IVANOV TO NOM||.

L'instruction EXAMINE permet de compter les apparitions d'un caractère déterminé dans une donnée élémentaire, de les remplacer par d'autres ou de repérer la position d'un caractère dans une donnée élémentaire. Deux formats de l'instruction EXAMINE sont possibles.

Dans sa première version, l'instruction EXAMINE contient:

1) le mot || EXAMINE|| suivi d'un nom-de-donnée élémentaire;

2) l'une des notations || TALLYING UNTIL FIRST|| TALLYING ALL|| TALLYING LEADING|| suivie d'un premier littéral non numérique à un seul caractère (littéral-1);

3) la notation || REPLACING BY|| suivi d'un deuxième littéral non numérique à un seul caractère (littéral-2). Cette clause est facultative.

Cette instruction examine la valeur de la donnée élémentaire et compte (selon l'option du point 2)): soit tous les caractères rencontrés avant la première apparition de littéral-1; soit tous les

caractères identiques à littéral-1 ; soit toutes les apparitions de littéral-1 avant la rencontre d'un caractère autre que littéral-1. Le nombre obtenu par le comptage est stocké dans le registre spécial nommé TALLY. Si, de plus, la clause REPLACING BY est spécifiée, tous les caractères comptés sont remplacés (au fur et à mesure du comptage) par le caractère du deuxième littéral. Le texte obtenu par suite des substitutions est considéré comme une nouvelle valeur de la donnée élémentaire nommée dans l'instruction.

Dans sa deuxième version, l'instruction EXAMINE contient:

1) le mot || EXAMINE || suivi d'un nom-de-donnée élémentaire;

2) l'une des notations || REPLACING ALL || REPLACING LEADING || REPLACING UNTIL FIRST || suivie d'un premier littéral non numérique à un caractère (littéral-1);

3) le mot || BY || suivi d'un deuxième littéral non numérique à un caractère (littéral-2).

Cette version de l'instruction EXAMINE ne diffère de la précédente que par le fait que le résultat du comptage de caractères n'est pas conservé.

Exemple 10.45.

|| EXAMINE PRIX REPLACING LEADING '0'BY'*' ||

|| EXAMINE MOT TALLYING ALL'A' ||

|| EXAMINE MOT TALLYING UNTIL FIRST'A'
REPLACING BY'␣' ||.

10.13.2. Instruction SORT. L'instruction SORT sert à décrire l'opération de tri des enregistrements qui se fait en fonction d'une ou plusieurs clés. Cette instruction contient:

1) le mot || SORT || suivi d'un nom-de-fichier;

2) l'une des notations || ON ASCENDING KEY || ON DESCENDING KEY || suivie d'un nom-de-donnée élémentaire; le texte de cette clause peut être répété plusieurs fois en changeant de nom-de-donnée;

3) le mot || GIVING || suivi d'un deuxième nom-de fichier; cette clause peut être absente, on suppose alors que le résultat du tri représente un nouveau contenu du fichier initial.

Exemple 10.46.

|| SORT FICH-5 ON ASCENDING KEY ANNEE
ON ASCENDING KEY SALAIRE ON DESCENDING KEY AGE ||.

§ 10.14. Instructions de l'appel des sous-programmes

10.14.1. Instructions CALL, EXIT PROGRAM. L instruction CALL est une suite formée par:

1) le mot || CALL|| suivi d'un littéral non numérique qui désigne un sous-programme ou un point d'entrée dans ce programme;

2) le mot || USING|| suivi d'une liste de noms-de-donnée. Ces derniers désignent les variables du programme principal servant dans le programme appelé. Cette clause est facultative.

Les données spécifiées dans la clause USING sont décrites dans la LINKAGE SECTION du programme appelé avec les niveaux 01 ou 77.

L'exécution d'une instruction CALL consiste en la transmission du contrôle au programme appelé (en chargeant préalablement celui-ci dans la mémoire s'il le faut).

Exemple 10.47.

|| CALL PR-C4||

|| CALL CORR USING A2 TABLE1||.

L'instruction EXIT PROGRAM représente les mots || EXIT PROGRAM||. Il faut que cette instruction soit l'unique instruction d'un paragraphe. Elle provoque, lors de l'exécution d'un sous-programme, le retour dans le programme principal à l'instruction suivant immédiatement l'instruction CALL. Si l'instruction EXIT PROGRAM appartient au programme principal, son exécution a pour effet le passage du contrôle au superviseur.

10.14.2. Instruction ENTRY. Elle est utilisée dans un sous-programme pour en identifier un point d'entrée. Elle représente le mot || ENTRY|| suivi d'un littéral, suivi facultativement du mot || USING|| suivi d'une liste de noms-de-donnée. Le littéral est le même que dans l'instruction CALL du programme principal.

La clause USING fait correspondre des variables du programme principal à certaines variables du sous-programme. Au premier nom-de-donnée de la clause USING de l'instruction CALL correspond le premier paramètre de la clause USING de l'instruction ENTRY et ainsi de suite jusqu'au dernier, le nombre de noms-de-donnée dans chacune des deux listes devant être identique.

§ 10.15. Instruction IF

L'instruction IF du COBOL est analogue aux instructions conditionnelles de l'ALGOL, mais plus simple.

Considérons deux versions de l'instruction IF.

Dans la première version, elle représente le mot || IF || suivi d'une expression logique, puis d'une instruction impérative ou des mots || NEXT SENTENCE ||. Elle provoque alors le calcul de la valeur de l'expression logique et exécution de l'instruction spécifiée (qui est vide, si l'option NEXT SENTENCE est choisie) dans le cas de « vrai ». Dans le cas de « faux » la commande passe à l'instruction suivant immédiatement l'instruction IF.

La deuxième version, qui est plus compliquée, représente le texte de la première version suivi du mot || ELSE ||, puis d'une deuxième instruction ou des mots || NEXT SENTENCE ||. Cette fois l'instruction IF provoque le calcul de la valeur de l'expression logique, puis l'exécution soit de la première instruction spécifiée (qui est vide si NEXT SENTENCE est choisi), soit de la deuxième instruction spécifiée (elle aussi est vide en cas de NEXT SENTENCE), selon que la valeur de l'expression logique s'avère « vrai » ou « faux ».

Si l'on désigne par P l'expression logique, par Q_1 et Q_2 la première et la deuxième instructions spécifiées, alors la première version de l'instruction IF peut être décrite en YALS comme suit:

$$P \underset{1}{\llcorner} Q_1 \underset{1}{\lrcorner},$$

et la deuxième par l'expression

$$P \underset{1}{\sqsubset} Q_1 \underset{2}{\sqsubset} \underset{1}{\lrcorner} Q_2 \underset{2}{\llcorner}.$$

EXEMPLE 10.48. La notion de l'instruction IF peut être illustrée par les exemples suivants:

	IF SALAIRE > 100 COMPUTE X = (Y + Z) * 0,25	
	IF X > Y ** 2 NEXT SENTENCE ELSE SUBTRACT X FROM Z	
	IF X > Y ** 2 ADD X Y TO Z ELSE SUBTRACT X FROM Y	

§ 10.16. Sémantique du COBOL

10.16.1. Particularités du langage COBOL. Une proposition achevée du COBOL est un programme COBOL. Nous savons déjà qu'il comprend quatre divisions. La division de l'identification est d'ordre « juridique », elle sert à identifier le programme et fournit tout autre commentaire le concernant. La division de l'environnement décrit l'ordinateur sur lequel le programme doit être compilé et exécuté, les unités d'entrée-sortie, etc.

La division des données décrit toutes les données qui sont traitées dans un programme COBOL, mais la description dépend de l'organisation des données. En effet, soit deux groupes de données

$$X = \{x_1, x_2, \ldots, x_n\}; \; Y = \{y_1, y_2, \ldots, y_m\}. \qquad (10.1)$$

Supposons qu'on effectue sur ces données une opération dont le résultat soit encore un groupe de données. Une telle opération peut être envisagée comme « globale », i.e. s'effectuant sur les données considérées chacune comme un tout. Alors toute l'information concernant cette opération doit être mise dans la machine et dans l'algorithme d'exécution du programme COBOL. Une autre voie est possible. Le résultat cherché peut être obtenu à l'aide d'une ou de plusieurs opérations « locales » qui s'effectuent sur les éléments de groupes. On a alors besoin d'une information sur la structure des groupes. C'est exactement l'information de ce genre que contient la division des données. Les opérations sur les groupes de données sont décrites dans la division de traitement comme des opérations locales sur leurs éléments.

Exemple 10.49. S'il y a les groupes de données (10.1), et que le résultat soit encore un groupe que nous désignons toujours par X :

$$X = \{x_1, x_2 + y_5, x_3, \ldots, x_n\},$$

alors, en considérant l'opération comme « globale », on peut écrire :

$$X := \omega(X, Y).$$

A l'aide d'une opération « locale », le même résultat se présente comme suit :

$$x_2 := x_2 + y_5.$$

Même la division de traitement contient des clauses qui servent non pas à déterminer l'algorithme mis dans le programme COBOL, mais à indiquer au compilateur comment organiser et réaliser le programme qu'on aura après assemblage.

Exemple 10.50. La division de traitement peut contenir l'instruction ||OPEN INPUT FICH-1 WITH NO REWIND|| (voir le p. 10.12.1). On en a besoin pour construire correctement le programme-objet, mais du point de vue de l'algorithme elle n'est pas indispensable. Tout de même, l'écriture d'une instruction READ (ou WRITE) sans un OPEN qui la précède doit être considérée comme une erreur de syntaxe.

10.16.2. Algorithme d'exécution d'un programme COBOL. La sémantique d'un langage algorithmique peut être donnée soit comme un algorithme d'assemblage, soit comme un algorithme d'exécution des algorithmes formulés dans ce langage (voir le p. 5.5.5). On dirait que le COBOL prévoit que sa sémantique soit donnée au moyen d'un algorithme d'assemblage. Les programmeurs expérimentés qui programment en COBOL distinguent nettement à travers le programme-source le programme-objet. Faute de place, nous ne décrivons ici l'algorithme d'exécution d'un programme COBOL que sous une forme très abrégée. L'effet de chacune des instructions est déjà suffisamment expliqué dans ce chapitre (bien que d'une manière non formelle). Donc nous nous bornerons à une brève description du processus global d'exécution d'un programme COBOL, une construction rigoureuse et complète de l'algorithme n'étant question que du temps et de l'application.

Si un état initial de la mémoire est donné (alors l'état de la mémoire extérieure doit être non vide, celui de la mémoire intérieure doit être vide), le processus d'exécution d'un programme COBOL se réduit à ce qui suit.

1°. Trouver la première instruction dans la PROCEDURE DIVISION. Exécuter 2°.

2°. Si l'instruction considérée est de l'un des types:

a) instruction arithmétique impérative,

b) instruction d'entrée-sortie (sauf READ),

c) instruction de transfert des données,

d) l'une des instructions ALTER, EXIT, ENTRY, FREE aller à 3°, si non, à 7°.

3°. Exécuter l'instruction considérée (en envisageant l'exécution d'un sous-programme comme une étape d'exécution de l'instruction ENTRY). Aller à 4°.

4°. Vérifier si la PROCEDURE DIVISION contient une instruction après celle exécutée. Si oui aller à 5°, si non, à 6°.

5°. Passer, dans la PROCEDURE DIVISION, à l'instruction suivante. Revenir à 2°.

6°. Fin.

7°. Si l'instruction considérée est un STOP, exécuter 8°, si non, 11°.

8°. Si l'instruction contient le mot RUN, exécuter 9°, si non, 10°.

9°. Ecrire le mot DISPATCHER et terminer le processus (qui se déroule sans résultat).

10°. Exécuter l'instruction, arrêter et attendre l'ordre de continuer le travail. Une fois l'ordre reçu, aller à 4°.

11°. Si l'instruction considérée est un PERFORM, aller à 12°, si non, à 15°.

12°. Si le groupe de procédures à exécuter n'est pas suivi immédiatement d'une instruction EXIT, aller à 13°, si non, à 14°.

13°. Rendre la commande à la procédure initiale du groupe de procédures à exécuter et reprendre la commande de la dernière de ces procédures tant que cela est demandé par la condition de l'instruction PERFORM. Ensuite aller à 4°.

14°. Procéder comme en 13°, à ceci près que l'exécution du groupe de procédures est aussi considérée comme terminée à chaque sortie de ce groupe. Après l'exécution du groupe, passer dans la PROCEDURE DIVISION à l'instruction EXIT qui suit immédiatement le groupe mentionné. Aller à 2°.

15°. Si l'instruction considérée est un GO TO aller à 16°, si non, à 17°.

16°. Déterminer le nom de la procédure suivante à exécuter et passer à sa première instruction. Aller à 2°.

17°. Si l'instruction considérée est un CALL, aller à 18°, si non, à 19°.

18°. Trouver le programme COBOL nommé dans l'instruction. Aller à 1°.

19°. Si l'instruction considérée est un EXIT PROGRAM, aller à 20°, si non, à 22°.

20°. Si c'est le programme principal qui s'exécute, aller à 9°, si c'est un programme « appelé », aller à 21°.

21°. Trouver dans le programme appelant l'instruction CALL qui a passé la commande au programme donné. La considérer. Aller à 4°.

22°. Si l'instruction considérée est arithmétique conditionnelle, aller à 23°., si non, à 26°.

23°. Effectuer l'opération principale de l'instruction arithmétique. Aller à 24°.

24°. Voir si un dépassement de capacité a lieu. Si oui, aller à 25°, si non, à 4°.

25°. Considérer l'instruction intérieure contenue dans l'instruction donnée. Aller à 2°.

26°. Si l'instruction considérée est un READ, aller à 27°, si non, à 29°.

27°. Effectuer l'opération principale de l'instruction. Aller à 28°.

28°. Si le fichier est épuisé, aller à 25°, si non, à 4°.

29°. Si l'instruction considérée est un IF dans sa première version, aller à 30°, si non, à 31°.

30°. Calculer la valeur de l'expression logique de l'instruction. En cas de « vrai » aller à 25°, en cas de « faux », à 4°.

31°. On considère la deuxième version de l'instruction IF. Calculer la valeur de son expression logique. En cas de « vrai » aller à 32°, en cas de « faux », à 34°.

32°. Lorsqu'on a les mots NEXT SENTENCE à la place de la première instruction intérieure, aller à 4°, si non, à 33°.

33°. Aborder la première instruction intérieure en la considérant comme occupant la place de la deuxième instruction intérieure. Aller à 2°.

34°. Si la place de la deuxième instruction intérieure est occupée par les mots NEXT SENTENCE, aller à 4°, si non, à 35°.

35°. Considérer la deuxième instruction intérieure. Revenir à 2°.

REMARQUE. Les opérations d'introduction de données de la mémoire extérieure dans la mémoire intérieure, le traitement et de sortie, utilisent, en plus de la PROCEDURE DIVISION, la DATA DIVISION qui contient la description de la structure des données.

BIBLIOGRAPHIE

1. Revised Report on the Algorithmic Language ALGOL 60. Edited by P. NAUR, International Federation for Information Processing, 1962.
2. Алгоритмический язык АЛГОЛ-68 (Langage algorithmique ALGOL-68) под общей редакцией А. П. Ершова, сб. « Кибернетика », изд-во АН УССР, Киев, 1970, вып. 1.
3. АЛЬФА-система автоматизации программирования (Système ALPHA de programmation automatisée) под ред. А. П. Ершова, Новосибирск, 1965.
4. БЕЛОКУРСКАЯ И. А., КУШНЕРЕВ Н. Т., НЕМЕНМАН М. Е.— Диспетчер ЭВМ « Минск-32 » (BELOKOURSKAYA I. A., KOUCHNEREV N. T., NEMENMAN M. E. — Dispatcher « Minsk-32 »), « Статистика », М., 1973.
5. БРУДНО А. Л. — Введение в программирование в содержательных обозначениях (BROUDNO A. L. — Initiation à la programmation dans les désignations non formelles), « Наука », 1965.
6. ВАСИЛЬЕВ В. А. — Язык АЛГОЛ-68. Основные понятия (VASSILIEV V. A. — Langage ALGOL 68. Notions de base), « Наука », 1972.
7. ВИЛЕНКИН С. Я., ТРАХТЕНГЕРЦ Э. А. — Математическое обеспечение управляющих вычислительных машин (VILENKINE S. Ya., TRAKHTENHERZ E. A. — Le software des ordinateurs de gestion), « Энергия », 1972.
8. HILBERT D., ACKERMANN W. — Grundzüge der theoretischen Logik, Berlin, 1938.
9. GINSBURG S. — The mathematical theory of context free languages, McGraw-Hill, 1966.
10. ГЛУШКОВ В. М. — Синтез цифровых автоматов (GLOUCHKOV V. M. — Synthèse des automates digitaux), Физматгиз, 1962.
11. GOODSTEIN R. L. — Mathematical Logic, Leicester univ. Press, 1957.
12. GERMAIN C. B. — Programming the IBM/360, Prentice-Hall inc., Englewood Cliffs, 1967.
13. ЕРШОВ А. П., ЛЯПУНОВ А. А. — О формализации понятия программы (ERCHOV A. P., LIAPOUNOV A. A. — Sur la formalisation de la notion de programme), сб. « Кибернетика », n° 5, Киев, 1967.
14. INGERMANN P. Z. — A syntax-oriented translator, Academic Press, 1966.
15. КАМЫНИН С. С., ЛЮБИМСКИЙ Э. З. — Алгоритмический машинно-ориентированный язык АЛМО (KAMYNINE S. S., LUBIMSKI E. S. — Langage algorithmique ALMO orienté vers la machine), сб. « Алгоритмические языки и алгоритмы », ВЦ АН СССР, М., 1967, вып. 1.
16. КИТОВ А. И. — Программирование информационно-логических задач (KITOV A. I. —Programmation des problèmes informatico-logiques), « Сов. радио », 1967.

17. КИТОВ А. И., КРИНИЦКИЙ Н. А. — Электронные вычислительные машины (KITOV A. I., KRINITSKI N. A. — Calculateurs électroniques), « Наука », 1965.
18. КИТОВ А. И., КРИНИЦКИЙ Н. А. — Электронные цифровые машины и программирование (KITOV A. I., KRINITSKI N. A. — Calculateurs électroniques digitaux et programmation), Физматгиз, 1961.
19. KLEENE S. C. — Introduction to Metamathematics, Van Nostrand, Princeton, 1952.
20. КОБРИНСКИЙ Н. Е., ТРАХТЕНБРОТ Б. А. — Введение в теорию конечных автоматов (KOBRINSKI N. E., TRACHTENBROT B. A. — Introduction à la théorie des automates finis), Физматгиз, 1962.
21. КОЛМОГОРОВ А. Н., УСПЕНСКИЙ В. А. — К определению алгоритма (KOLMOGOROV A. N., OUSPENSKI V. A. — Sur la définition de l'algorithme), УМН, 1958, т. XIII, вып. 4 (82).
22. КОЛМОГОРОВ А. Н. — Предисловие редактора перевода (KOLMOGOROV A. N. — Préface à la traduction russe du livre : PETER R. — Rekursive Funktionen), 1954.
24. КРИНИЦКИЙ Н. А. — Логическая операторная схема ЭЦМ (KRINITSKI N. A. — Schéma logique opératoriel d'un ordinateur), Энциклопедия « Автоматизация производства и промышленная электроника », т. 2, « Сов. Энциклопедия », 1963.
25. КРИНИЦКИЙ Н. А. — О некоторых неточностях АЛГОЛа-60 (KRINITSKI N. A. — Sur quelques imperfections de l'ALGOL 60), сб. « Цифровая вычислительная техника и программирование », « Сов. радио », № 5, 1969.
26. КРИНИЦКИЙ Н. А. — Операция машинная (KRINITSKI N. A. — Opération de machine), Энциклопедия « Автоматизация производства и промышленная электроника », т. 2, « Сов. Энцикл. », 1963.
27. КРИНИЦКИЙ Н. А. — Равносильные преобразования и программирование (KRINITSKI N. A. — Transformations équivalentes et programmation), « Сов. радио », 1970.
28. КРИНИЦКИЙ Н. А. — О некоторых формальных языках (KRINITSKI N. A. — Sur quelques langages formels), сб. « Цифровая вычислит. техника и программирование ». « Сов. радио », № 7.
29. КРИНИЦКИЙ Н. А. — Язык логических схем (KRINITSKI N. A. — Langage des schémas logiques), сб. « Цифровая вычислит. техника и программирование », « Сов. радио », № 1, 1966.
30. КРИНИЦКИЙ Н. А. — Язык сетевых схем (KRINITSKI N. A. — Langage des réseaux), сб. « Цифровая вычислит. техника и программирование », « Сов. радио », № 3, 1967.
31. КРИНИЦКИЙ Н. А., МИРОНОВ Г. А., ФРОЛОВ Г. Д. — Описание системы команд с помощью элементарных машинных операций (KRINITSKI N. A., MIRONOV G. A., FROLOV G. D. — Description des instructions de machine à l'aide d'opérations élémentaires de machine), сб. « Цифровая вычислит. техника и программирование », « Сов. радио », № 4, 1968.
32. КРИНИЦКИЙ Н. А., МИРОНОВ Г. А., ФРОЛОВ Г. Д. — Программирование (KRINITSKI N. A., MIRONOV G. A., FROLOV G. D. — Programmation), изд. 2-е « Наука », 1966.
33. КРИНИЦКИЙ Н. А., МИРОНОВ Г. А., ФРОЛОВ Г. Д. — Формальное определение некоторого класса сложных систем (KRINITSKI N. A., MIRONOV G. A., FROLOV G. D. — Une définition formelle d'une classe de systèmes complexes), сб. « Большие системы », « Наука », 1971.
34. КРИНИЦКИЙ Н. А., ПРОСКУРОВ В. С. — Структура математического обеспечения АСПР (KRINITSKI N. A., PROSKOUROV V. S. — Structure

du software de ASPR), сб. « Вопросы математического обеспечения ЭВМ и систем расчетов », Госплан СССР, вып. 5, 1973.
35. КУЛАКОВСКАЯ В. П., РОМАНОВСКАЯ Л. М., САВЧЕНКО Т. А., ФЕЛЬДМАН Л. С. — КОБОЛ ЭВМ « Минск-32 » (KOULAKOVSKAIA V. P., ROMANOVSKAIA L. M., SAVTCHENKO T. A., FELDMAN L. S. — COBOL Minsk-32), « Статистика », 1973.
36. КУРОШ А. Г. — Лекции по общей алгебре (KOUROSH A. G. — Leçons sur l'algèbre générale), « Наука », 1973.
37. КУШНЕРЕВ Н. Т., НЕМЕНМАН М. Е., ЦЕГЕЛЬСКИЙ В. И. — Программирование для ЭВМ « Минск-32 » (KOUCHNEREV N. T., NEMENMAN M. E., TSEGUELSKI V. I. — Programmation pour l'ordinateur « Minsk-32 »), « Статистика », 1973.
38. ЛАВРОВ С. С. — Универсальный язык программирования (LAVROV S. S. — Langage de programmation universel), « Наука », 1964.
39. ЛАВРОВ С. С. — Введение в программирование (LAVROV S. S. — Introduction à la programmation), « Наука », 1973.
40. ЛАВРОВ С. С., ГОНЧАРОВА Л. И. — Автоматическая обработка данных (LAVROV S. S., GONTCHAROVA L. I. — Traitement automatique des données), « Наука », 1971.
41. LEDLEY R. — Programming and Utilising Digital Computers, New-York, 1962.
42. ЛЯПУНОВ А. А. — О логических схемах программ (LIAPOUNOV A. A.— Sur les schémas logiques de programmes), сб. « Проблемы кибернетики », № 1, « Наука », 1968.
43. ЛЯПУНОВ А. А. — К алгебраической трактовке программирования (LIAPOUNOV A. A. — Sur l'interprétation algébrique de la programmation), сб. « Проблемы кибернетики », № 8, 1959.
44. ЛЯПУНОВ А. А. — О некоторых общих вопросах кибернетики (LIAPOUNOV A. A. — Sur quelques problèmes généraux de la cybernétique), сб. « Проблемы кибернетики », № 1, 1958.
45. МАЛЬЦЕВ А. И. — Алгоритмы и рекурсивные функции (MALTSEV A. I. — Algorithmes et fonctions récursives), « Наука », 1965.
46. МАРКОВ А. А. — Теория алгорифмов (MARKOV A. A. — Théorie des algorithmes), Труды Матем.ин-та им. Стеклова, т. XII, изд-во АН СССР, 1954.
47. MARKOWITZ H. M., HAUZNER B., KARR H. W. — SIMSCRIPT — A simulation programming language, Prentice-Hall inc., 1963.
48. Мультипрограммирование и разделение времени (La multiprogrammation et le temps partagé), Recueil d'articles, Ed. Mir. 1970.
49. Мультипроцессорные вычислительные системы (Systèmes calculateurs à plusieurs processeurs), под ред. ХЕТАГУРОВА Я. А., « Энергия », 1971.
50. НОВИКОВ П. С. — Элементы математической логики (NOVIKOV P. S. — Eléments de la logique mathématique), Физматгиз, 1959.
51. ROGERS H. — Theory of recursive functions and effective computability, MIT, Cambridge, Mass., 1956.
52. SAXON J. A. — COBOL, Prentice-Hall, 1963.
53. Системное и теоретическое программирование (Programmation des systèmes et programmation théorique), сб. трудов под ред. КОТОВА В. Е., СО АН СССР, Новосибирск, 1972.
54. Современное программирование (Programmation moderne), сб. статей (traduction de l'anglais), « Сов. радио », т. 1, 1966, т. 2, 1967.
55. ТРАХТЕНБРОТ Б. А. — Алгоритмы и машинное решение задач (TRACHTENBROT B. A. — Algorithmes et résolution de problèmes sur calculateurs), изд. 2-е, « Наука », 1960.
56. WALSH D. A. — A guide for software documentation, ACT, 1969.
57. WILKES M. V. — Time-sharing Computer Systems, Macdonald, 1968.

58. Универсальный язык программирования PL/1 (Langage de programmation universel PL/1), под ред. КУРОЧКИНА В. М., « Мир », 1968.
59. УСПЕНСКИЙ В. А. — Лекции о вычислимых функциях (OUSPENSKI V. A. — Lecons sur les fonctions calculables), Физматгиз, 1960.
60. FLORES I. — Computer Software, Prentice-Hall, 1965.
61. ФРОЛОВ Г. Д. — Некоторые равносильные преобразования циклов в логической схеме (FROLOV G. D. — transformations équivalentes des boucles dans un schéma logique), сб. « Цифровая вычислит. техника и программирование », « Сов. радио », № 3, 1967.
62. HOPGOOD F. R. A. — Compiling Techniques, London, New York, 1970.
63. CHURCH A. — Introduction to Mathematical Logic, Princeton, 1956.
64. ШИЛЛЕР Ф. Ф. — Алгоритмический язык описания экономико-математических задач — АЛГЭМ (SCHILLER F. F. — Langage algorithmique), сб. « Цифровая вычислит. техника и программирование », « Сов. радио », № 1, 1966.
65. EBBINGHAUS H.-D., MAHN F.-K. — Turing Maschinen und berechenbare Funktionen, Selecta Mathematica II, Berlin, Heidelberg, New York, 1970.
66. ASHBY W. ROSS — An introduction to cybernetics, London, 1956.
67. ЮЩЕНКО Е. Л., БАБЕНКО Л. П., МАШБИЦ Е. И. — КОБОЛ (YOUCHTCHENKO E. L., BABENKO L. P., MACHBITS E. I. (éditeurs) — COBOL), « Вища школа », Киев, 1973.
68. Языки программирования (Langages de programmation), сб. статей, перевод с англ. под ред. КУРОЧКИНА В. М. (traduction de l'anglais revue par KOUROTCHKINE V. M.), « Мир », 1972.
69. ЯНОВ Ю. И. — О логических схемах алгоритмов (IANOV Y. I. — Sur les schémas logiques d'algorithmes), сб. « Проблемы кибернетики », № 1, 1958.
70. BOLLIET L., GASTINEL N., LAURENT P. J. — Un nouveau langage scientifique ALGOL. Manuel pratique, Hermann, 1964.

INDEX ALPHABÉTIQUE DES MATIÈRES

ALGOL (274-307)

COBOL (443-494)

FORTRAN (308-355)

PL/1 (356-442)

YALS (74-120)

TABLE DES MATIÈRES

A NOS LECTEURS

Les Editions Mir vous seraient très reconnaissantes de bien vouloir leur communiquer votre opinion sur le contenu de ce livre, sa traduction et sa présentation, ainsi que toute autre suggestion.
Notre adresse:
Editions Mir, 2, Pervi Rijski péréoulok,
Moscou, I-110, GSP, U.R.S.S.

Imprimé en Union Soviétique